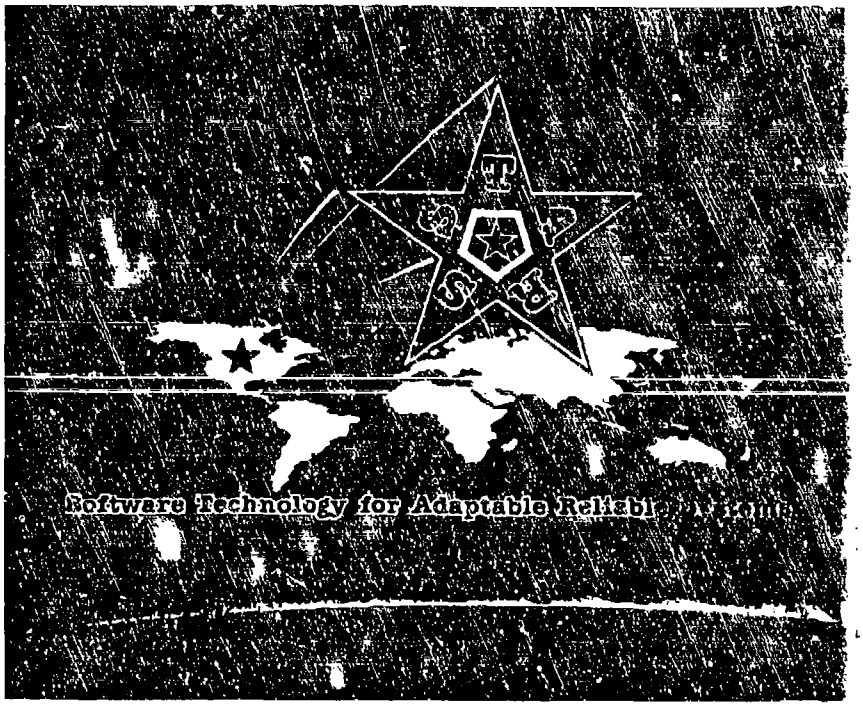
TASK: UR40
CDRL: 01210

(2)

UR40 — Repository Integration AdaTAU Software User's Manual

Informal Technical Data

UNISYS



STARS-RC-01210/003/00

26 October 1990

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	26 October 1990	Final

4. TITLE AND SUBTITLE
Reusability Library Framework (RLF) Librarian User's Manual

5. FUNDING NUMBERS
STARS Contract
F19628-88-D-0031

6. AUTHOR(S)
James J. Solderitsch
Ray McDowell

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Unisys Corporation
12010 Sunrise Valley Drive
Reston, VA 22091

8. PERFORMING ORGANIZATION REPORT NUMBER
GR-7670-1169(NP)

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Department of the Air Force
Headquarters, Electronic Systems Division (AFSC)
Hanscom AFB, MA 01731-5000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
01210 Volume III

11. SUPPLEMENTARY NOTES
There are two other related Reusability Library Framework (RLF) reports:
(RLF) AdaTau User's Manual and (RLF) AdaKNET User's Manual

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release;
distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

While librarians are not the only sorts of applications that have been built with the RLF, they are the applications that motivated the initial development of the RLF. This manual describes a demonstration librarian built for the domain of Ada benchmark programs. This application is typical of many RLF applications in that it uses a hybrid knowledge representation system incorporating an integrated form of AdaTAU and AdaKNET. The manual provides a librarian system overview and provides an annotated sample usage session. The manual also presents the hybrid knowledge base description language used to connect AdaKNET and AdaTAU.

14. SUBJECT TERMS
Librarian System Overview
Librarian System Components
Using the Librarian

15. NUMBER OF PAGES
72

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

TASK: UR40
CDRL: 01210
26 October 1990

INFORMAL TECHNICAL REPORT

For The

SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS (STARS)

Reusability Library Framework (RLF)
AdaTAU Software
User's Manual

STARS-RC-01210/003/00
Publication No. GR-7670-1171(NP)
26 October 1990

Data Type: A005, Informal Technical Data

CONTRACT NO. F19628-88-D-0031
Delivery Order 0002

Prepared for:

Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

Prepared by:

Unisys Defense Systems
Tactical Systems Division
12010 Sunrise Valley Drive
Reston, VA 22091

TASK: UR40
CDRL: 01210
26 October 1990

INFORMAL TECHNICAL REPORT

For The

SOFTWARE TECHNOLOGY FOR ADAPTABLE, RELIABLE SYSTEMS
(STARS)

Reusability Library Framework (RLF)
AdaTAU Software
User's Manual

STARS-RC-01210/003/00
Publication No. GR-7670-1171(NP)
26 October 1990

Data Type: A005, Informal Technical Data

CONTRACT NO. F19628-88-D-0031
Delivery Order 0002

Prepared for:

Electronic Systems Division
Air Force Systems Command, USAF
Hanscom AFB, MA 01731-5000

Prepared by:

Unisys Defense Systems
Tactical Systems Division
12010 Sunrise Valley Drive
Reston, VA 22091

PREFACE

This document was prepared by Unisys Defense Systems, Valley Forge Operations, in support of the Unisys STARS Prime contract under the Repository Integration task (UR40). This CDRL, 01210, Volume III, is type A005 (Informal Technical Data) and is entitled "AdaTAU Software User's Manual".

This document has been reviewed and approved by the following Unisys personnel:

UR40 Task Manager: Richard E. Creps

Reviewed by: Richard E. Creps FOR
Teri F. Payton, System Architect

Approved by: Hans W. Polzer
Hans W. Polzer, Program Manager

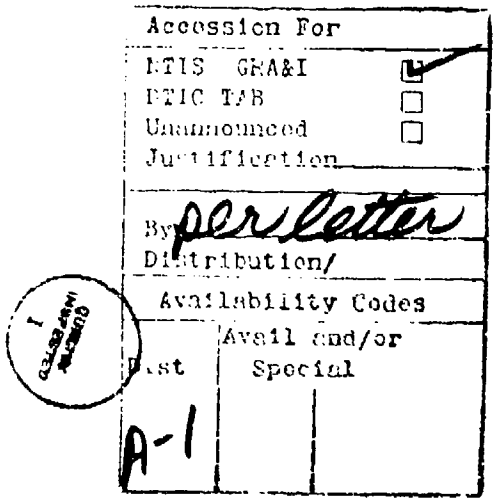


Table of Contents

1. Scope 1
1.1. Identification 1
1.2. Purpose 1
1.3. Introduction 2
2. Referenced Documents 3
3. AdaTAU System Overview 4
3.1. Basic Architectural Features 4
3.2. Fundamental Architectural Elements 8
4. AdaTAU System Components 22
4.1. Facts 25
4.2. Fact Lists 27
4.3. Fact Value Lists 28
4.4. Fact Schemas 29
4.5. Fact Base Schemas 31
4.6. Fact Bases 33
4.7. Fact Parameters 35
4.8. Fact Parameter Lists 37
4.9. IRules 38
4.10. FRules 40
4.11. QRules 43
4.12. Questions 45
4.13. Response Schemas 47
4.14. Rule Bases 50
4.15. Agendas 52
4.16. Basic AdaTAU Configuration 54
4.17. Generic Advanced AdaTAU Configuration 56
4.18. RLF Instance of Advanced AdaTAU Configuration 59
4.19. Component Persistance Management 60
4.20. Basic AdaTAU Persistance Management 62
4.21. Distributed AdaTAU Persistance Management 63
4.22. Reverse RBDL Translator 64
4.23. AdaTAU Inference Cycle Components 65
4.24. Distributed AdaTAU Inference Cycle Components 66
4.25. AdaTAU Basic Inference 67
4.26. Distributed AdaTAU Inference 68
5. AdaTAU Specification Language — RBDL 69
6. Using AdaTAU 72
6.1. Creating AdaTAU Knowledge Bases 73
6.2. Building an AdaTAU Application 74
6.3. Sample Session 75

7. Notes 81
7.1. Facts 81
7.2. Rules 81
7.3. Inference 82
7.4. GLOSSARY 84

Appendices

A. RBDL Syntax and Summary A-1
A.1. Extended BNF (EBNF) Meta-Symbols 1
A.2. RBDL EBNF and Semantics 1
A.2.1. AdaTAU Specification 1
A.2.2. Local Inferencer Definition 2
A.2.3. Fact Base Schema Definition 3
A.2.4. Factbase Definition 4
A.2.5. Irulebase Definition 4
A.2.6. Questionbase Definition 5
A.2.7. Qrulebase Definition 6
A.2.8. Frulebase Definition 6
A.2.9. Inferencer Definition 7
A.2.10. Fact 7
A.2.11. Lexical Elements 8
A.3. RBDL EBNF Syntax Summary 8
B. RBDL Extended Example B-1

References

Table of Figures

Figure 1. AdaTAU Layered Abstractions 7
Figure 2. Basic AdaTAU Inference 15
Figure 3. Distributed AdaTAU Inference 17
Figure 4. Basic AdaTAU Component Taxonomy 23
Figure 5. Creating an AdaTAU Application 72

1. Scope

This document assumes that the user has a basic understanding of the Ada language and wishes to learn how to incorporate knowledge-based capabilities into a larger system. This document is not tutorial in nature with regard to the Ada language, nor does it cover basic material from the field of Artificial Intelligence (AI). In fact, some of the fundamental features of the system described in this manual are based on ideas described in the AI literature. The interested reader is referred to one of the many texts on Ada or AI; in particular, the Ada Language Reference Manual [LRM83] and *The Handbook of Artificial Intelligence*, Volume 1[Barr81].

1.1. Identification

This Software User's Manual provides a description of the content and basic operating procedures of AdaTAU, a subsystem level component of the Reusability Library Framework (RLF). Other major components of the RLF include AdaKNET, and the Librarian application, which are covered in separate user's manuals. AdaTAU provides knowledge representation and inferencing capabilities via rule and fact base abstractions, and an associated control strategy that supports the extension of fact bases following the application of rules drawn from the rule bases. AdaTAU is made up of various packages providing Abstract Data Types (ADTs) that form the basis for the collection of services and objects provided within AdaTAU. This manual describes the individual package level components, as well as the major operations and objects defined within each component.

1.2. Purpose

The purpose of AdaTAU is to provide a rule-based knowledge representation capability within the RLF and to serve as a stand-alone subsystem that can be incorporated into larger Ada systems which require a rule-based component. Rule bases provide a declarative form of knowledge (or heuristics) that human "experts" use to make decisions within a current knowledge context. Such rules can be used to manage volatile information and make decisions consequent to this information which is garnered during the processing of other external data structures or through general interaction with a user. One example of such an external data structure is a semantic network such as the ones provided by the AdaKNET subsystem of the RLF. This static information is supplemented by information recorded as simple facts that are collected into fact bases. The current version of AdaTAU stores facts as simple attribute-value pairs.

Facts are used as input values to collections of rules organized into rule bases by AdaTAU. Rules whose input facts are all noted to occur within the current fact base will be "fired" with the result that new resultant facts can be added to the fact base and old facts can be removed. AdaTAU both maintains a collection of facts, and manages a collection of rules, through which information can be passed to an application which is processing its own data. AdaTAU can also direct a sequence of interactions with the user by posing questions to the user and receiving answers which cause changes to an AdaTAU fact base. In addition, this version of AdaTAU supports the partitioning of rule bases into focused "inference contexts" and includes rules that direct inference focus to the proper context.

1.3. Introduction

The remainder of this document is organized as follows. Section 2 lists a few particular reference works that have particular relevance to this document. Section 3 contains an overview of the AdaTAU system including separate views of the basic architectural features of AdaTAU and the fundamental elements of AdaTAU that contribute to these features. The former view presents a coarser-grained look at the AdaTAU system useful for Ada programmers who have some familiarity with rule-based systems and therefore can read the underlying Ada package specifications directly. The latter view provides more detail and explanation for those who are less experienced with Ada and knowledge-based programming. In section 4, a complete treatment of the AdaTAU package level components is given. Within each subsection detailing an individual package, the collection of basic objects, types and operations that make up the package are all covered separately. Section 5 presents a description of the knowledge base declaration language used to describe schemas for individual fact and rule bases and state facts that initialize particular fact bases. A hands-on view of AdaTAU, covering the steps necessary to integrate AdaTAU with other subsystems, is presented in section 6 including the use of AdaTAU's static description language to tailor AdaTAU to a particular application domain. Finally, section 7 provides some general background information about the concepts and terms used in this document. Section 7 also includes a glossary of important terms, acronyms and abbreviations.

2. Referenced Documents

In addition to the Ada LRM, and the AI Handbook referenced earlier, the following documents are useful as references in conjunction with this document. Documents marked with an asterisk (*) were delivered to the Naval Research Laboratory as part of the original STARS Foundation contract (number N00014-88-C-2052) that supported the initial development of the RLF.

(*) Reusability Library Framework AdaKNET/AdaTAU Design Report.

(*) Gadfly User's Manual.

AdaKNET User's Manual.

Librarian User's Manual.

The RLF Librarian: A Reusability Librarian Based on Cooperating Knowledge-Based Systems [McDowell89].

The Growing of an Organon: A Hybrid Knowledge-Based Technology and Methodology for Software Reuse [Simos88].

Construction of Knowledge-Based Components and Applications in Ada [Wallnau88].

Constructing Domain-Specific Ada Reuse Libraries [Solderitsch?].

3. AdaTAU System Overview

AdaTAU is based on TAU, a Unisys-proprietary production rule-based system that incorporates an agenda mechanism for directing interaction with a user along with a forward-chaining inference system. Another agenda mechanism is used to manage inference over a distributed collection of "inference bases" where the inference process spans the collection. Rule base systems provide a deductive capability to generate new information based on information already present in the system. Information is stored in the form of facts that are represented in AdaTAU as attribute, value pairs. Rule bases are simply collections of rules, each of which includes a list of facts that must be true to apply the rule, a list of facts to be concluded and/or a list of facts to be retracted when the rule is applied. Note that one or the other (but not both) of these lists could be empty.

Rule base systems have traditionally been written in Lisp and Prolog. However, AdaTAU demonstrates that Ada with its strong notions of data typing, data abstraction, and exception handling is a viable and efficient language system that supports this form of knowledge representation and processing. Ada will allow this component to be naturally integrated into larger knowledge processing systems that have need of a rule base component. The underlying pieces of AdaTAU are *facts* (collected into fact bases), and *rules* (collected into rule bases). Rules depend on facts (rule antecedents) and are used to derive new facts or remove old facts (rule consequents). Through Ada language features, we are able to separate fact bases from the rule bases that modify them. In addition, AdaTAU provides for schema definitions for both facts and rules that naturally extend the native type checking of Ada.

3.1. Basic Architectural Features

TAU is an acronym made up of the first letters of the phrase that summarizes the organization of this component — *Think, Ask, Update*. *Think* refers to the analysis of the fact base which is used to record information about the domain under consideration, and to the application of rules which directly modify this fact base (add or delete facts for example). Rules considered during this phase may also lead to the scheduling of queries which will be processed subsequently. *Ask* denotes the capability of posing questions, and recording responses in answer to the questions, that are scheduled as a result of the Think phase. Finally, the *Update* phase will modify the fact base in a manner that depends on the responses recorded in the Ask phase. The part of AdaTAU executing this basic inference cycle is called the *investigator*.

Distributed AdaTAU (DTAU) is an extension of basic AdaTAU (identified as centralized TAU (or just TAU) in comparison to DTAU) that provides for the localization of individual rule and fact bases into separate inference contexts which communicate with each other through fact parameters. The basic TAU investigator cycle is augmented with an additional phase that processes rules which help identify possible inference focus switches to more useful contexts. The importing application makes use of a focus evaluation and TAU invocation cycle to perform inference over the available inference contexts.

Some basic architectural features of AdaTAU are summarized in the following labeled paragraphs. The next section of this manual provides additional background on

these features.

Fact Base Schemas. AdaTAU provides an attribute-value structure for facts. Facts are thus simply viewed as pairs of properties and values of such properties. However, AdaTAU manages the relationship between particular properties and the permitted value sets that contain values used in particular facts pertaining to the property. AdaTAU defines a set of *fact base schemas* for each application. These schemas restrict the form and value sets for facts within a particular fact base.

Rule Bases. Rather than providing a single kind of rule base, three kinds of rules, each contained in a corresponding rule base, and each supporting a different kind of forward-chained inference processing relative to a common fact base, have been provided. Other kinds of rules and rule bases may be added in future versions of AdaTAU. For example, action rules could be defined so application operations (actions) could be invoked with modification of the fact base depending on the results of the operation. Such a mechanism would enable the application to control and interact with the TAU process.

An *IRule* (Inference Rule) is a rule which directly affects the fact base, and requires no input from the user. IRules are the direct realization of *if — then* kinds of rules.

A *QRule* (Question-asking Rule) is a rule which involves the eventual processing of user input. Depending on a response to a particular question associated with the rule, other facts can be added to, or deleted from, the fact base. However, before the question is even posed to the user, the antecedent facts of the QRule must be present in the fact base. AdaTAU separates the scheduling of the question and the asking of the question and provides for the ranking of the question numerically when it is inserted on the agenda of questions to be asked. In this way the user is presented with the most important question first.

An *FRule* (Focus-suggestion Rule) is a rule that identifies an inference context (typically by an application-specific name) where it is likely that the goal of the current inference process will be served (i.e., additional facts can be deduced). An FRule does not itself derive any new facts, directly or indirectly. When an FRule's antecedent facts are present in the fact base, the FRule will cause the focus agenda to be modified. Depending on the weight attached to a focus suggestion, an FRule may lead to the suspension of inference within the current context, or it may direct attention to an alternate context when no further inference progress is possible in the current context.

FRules are required only for DTAU. However, through Ada we are able to define a single rule base data type, and separate rule data types corresponding to each of the kinds of rules listed above. Rule bases are defined using the generic construct of Ada, and the individual rule bases are defined via instantiations of this generic with each individual rule type.

Agendas. An *agenda* is simply a prioritized queue of items where retrievals from the agenda are based on an agenda item's priority or weight. In AdaTAU, interactions with the user which are scheduled as a result of the firing of QRules are placed within a separate agenda that is consulted as AdaTAU executes. Each individual inference context is equipped with its own local agenda. In DTAU, focus switch suggestions result from the firing of FRules, and these are merged with the current contents of the global focus agenda. QRules post a *question* item to a local question agenda which the investigator portion of AdaTAU uses to interact with the user in an organized manner. The

question agenda, and questions retrieved from it, is processed during the Ask and Update phases of TAU. DTAU includes a separate *Think-Again* phase during which IRules and FRules are processed in advance of considering a focus switch.

Questions themselves refer to additional schema definitions called *response schemas* which package the actual text of the question to be asked of the user along with the permitted response type that AdaTAU will expect in answer to the question, and the corresponding fact base modifications attached to each response. AdaTAU checks to see that included facts conform to the fact base schemas established for each attribute-value pair. Analogously to the case for rules, the concept of agenda is implemented as an Ada generic package which is instantiated with an object type representing a particular agenda possibility, questions. Later versions of AdaTAU will make use of other instances of the agenda generic.

AdaTAU Layering. In designing and implementing AdaTAU, a layered abstraction approach has been taken which results in an onion-skin like view (figure 1) of the structure of AdaTAU. Two onion-skin views are provided for centalized and distributed AdaTAU. By layering the basic services and objects that are required in support of AdaTAU, we reduce visibility of the underlying support operations that are used internally by AdaTAU, and promote and encapsulate the basic higher level operations required to use AdaTAU effectively. The core of the onion represents the kernel operations upon which AdaTAU is based. This kernel includes the basic definitions for facts, fact bases, fact base schemas, rules (of the various types), rule bases and agendas. Each basic object and operations on that object are captured in a single Ada package that implements the objects as a well-defined abstract data type.

The next layer in the onion-skin picture provides the programmatic interface to the core features of AdaTAU and is realized as a single package called basic_configuration. This layer also provides some basic composite operations that relate two or more objects which are defined in different Ada packages. All of the essential capabilities of AdaTAU that an application is likely to require are made available through this interface. The application need not be concerned with any underlying definitions or implementation details.

The outer layer of the onion denotes a package that contains the basic operations required to implement a TAU-style inference capability from the base components constructed in the inner layers. An application built using this outer interface is guaranteed to make proper use of AdaTAU operations, and to be completely independent of any implementation decisions regarding the basic operations, or any composites that directly depend on these operations. In particular, any decisions made about the storage methodology for collections of basic objects (such as fact and rule bases) are irrelevant as long as an application utilizes AdaTAU through this outer interface. However, an application can choose to use the services of AdaTAU directly as provided in the basic_configuration package. An alternative inference strategy to the TAU model can be defined from these middle level operations. At the lowest level, an application designer can decide to use the basic operations directly, perhaps providing a family of new composite operations in the process.

In the case of distributed AdaTAU, an additional layer is introduced between the application layer and the basic TAU configuration. This layer provides the FRule

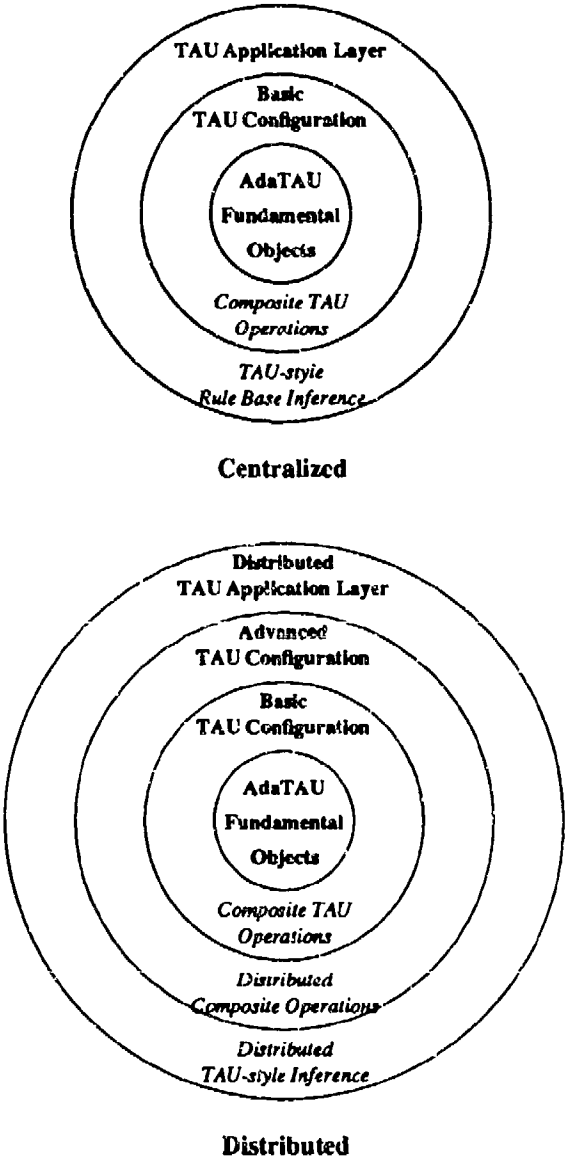


Figure 1. AdaTAU Layered Abstractions

mechanism and basic support for distributed operations and application integration. The additional layer makes use of the basic programmatic interface, but is not incorporated into the basic interface because the extra definitions and operations are not required for

the single inference base version. The application is free to provide the means by which individual TAU contexts are distributed. This layer is implemented in terms of an Ada generic package that accepts a basic Ada type and procedure as generic parameters. The procedure, supplied by the application designer, must produce an inference context based on particular type values for the type that instantiates the generic type parameter.

3.2. Fundamental Architectural Elements

Information that is managed by AdaTAU comes in two principal forms. Facts are units of data that represent features and properties of a domain. AdaTAU relies entirely on facts to capture the *what* of this domain. Rules, on the other hand, provide a mechanism through which facts can be either added to the collection (the fact base) being managed by AdaTAU, or deleted from this collection. Rules are thus agents of change in regard to the fact base. In the next subsections, the basic properties of facts, rules and related structures pertaining to their realization in Ada are discussed. Some additional background information about facts and rules can be found in section 7.

Facts

Facts are implemented as <attribute, value> pairs (a-v pairs). A simple example is <operation_status, prototype>. An attribute can be understood to be simply the name of a property of the domain under consideration. A value for an attribute provides a characterization of that property. Both attributes and values are implemented simply as strings, although AdaTAU provides a significant management component so that instances of these objects can be restricted and checked for conformance to declared rules for a-v pairs. Functions are provided to convert attributes and values to strings and vice versa. This allows the application using the facts package to print and manipulate attributes and values as strings and to create facts from strings. Two other routines, *Get_Attr* and *Get_Value* are provided to extract the attribute and value, respectively, from a fact.

Fact Lists

Simple collections of facts as manipulable structures are required in several places in the design of AdaTAU (e.g., as premise lists and conclusion lists for rules). We therefore provide a simple fact list structure by instantiating a generic list data structure, Tau_List, with the type fact. All the routines from the Tau_Lists package can be applied to fact lists. In addition, the routine *Search_Value* is provided which returns the value associated with an attribute of a fact in a fact list.

Fact Bases

Fact bases, viewed in their simplest form, are sets of a-v pairs. For a given domain, the list of possible attribute values is a finite set. For each attribute, AdaTAU permits either a single a-v pair, several distinct a-v pairs (with values restricted to a finite list), or an unspecified number of a-v pairs (with arbitrary values) to exist simultaneously in the fact base. These restrictions are encapsulated in a fact base schema, described below, which is part of the fact base data structure.

Fact bases can be created using *Create*, then filled with facts using the *Install* routine. Facts can also be deleted from the fact base using *Delete*. Other operations that

result in the creation of new fact bases are the set operations *Union*, *Intersection*, and *Difference*.

Several routines are provided to examine fact bases. A fact base's schema is returned using *Get_Schema*. There are two versions of *Search_Fact*. One returns a fact corresponding to a given attribute while the other returns a boolean value indicating whether or not a given fact is in the fact base. Similarly, *Search_Facts* returns a list of facts with a given attribute. Furthermore, the function *Dump_Base* allows a fact base to be converted into a fact list, which can be examined more easily. A fact base can also be tested for emptiness with the function *Empty*. The function *Subset* compares two fact bases to determine if one is a subset of the other. The function *Valid* can be used to test whether or not a fact is consistent with a fact base and the facts it already contains. Finally, the two functions *Compatible* and *Consistent* compare two fact bases to determine if they have equivalent fact base schemas and if their facts are consistent with each other. These functions can be used to determine if the two fact bases can be combined using set operations.

Fact and Fact Base Schemas

Fact schemas and fact base schemas provide a means of configuring the possible restrictions for a-v pairs that pertain to the various attributes. In particular, a fact schema restricts the values that can be paired with a given attribute. A simple representation of a fact schema is given by:

```
structure_type : one_of (stack, queue, string, deques,
                         rings, maps, sets, bags,
                         lists, trees, graphs, other);
```

In this example, structure_type is an attribute that can have exactly one of the values listed; thus, there can be at most one fact with this attribute within a fact base at any given time. In general, each distinct attribute has an associated fact type which is restricted to be either *one_of*, *some_of*, *any* or *reference*. A fact type of *one_of* or *some_of* specifies whether facts based on the attribute can take on only one, or several, values chosen from a finite list of possible values. In the latter case, a fact attribute may appear more than once in a fact base with a different value. A fact type of *any* specifies that facts based on the attribute can take on any arbitrary value. The fact type *reference* indicates that the fact value is actually a name that is a reference to a value description that is external to the fact itself. Reference facts are used when a fact value is too large to be embedded in the a-v pair and the value contents are typically placed within a file stored on disk. For each distinct attribute, a *Fact Schema* is maintained to capture this information for that attribute, as well as a list of allowable values (if the attribute is *one_of* or *some_of*). Routines are provided to create fact schemas and to extract the various components of a fact schema (*Get_Attr*, *Get_Type*, and *Get_Values*). As well, a function *Valid* is provided which returns true if a given fact is valid with respect to a given fact schema.

A fact base schema restricts the structure of the collection of facts that can belong to a fact base. Fact base schemas are implemented as lists of fact schemas, one for each attribute allowed in a fact base. Every fact base must have an associated fact base schema. *Create_Fact_Base_Schema* creates an empty fact base schema which can be

filled with fact schemas using *Add_Schema*. A routine, *Dump_Schema*, is provided to convert the fact base schema to a list of fact schemas, which can then be examined using routines from the generic package Tau_Lists. A fact base schema can be tested for emptiness using *Empty*. Several versions of the function *Valid* are provided to check for inconsistencies between fact base schemas and attributes, facts, fact lists, or fact schemas. Also, the fact schema corresponding to a particular attribute can be obtained using *Get_Schema*.

IRules

IRules implement an *if — then* sort of rule and are the simplest kind of rule to process. They essentially encapsulate two different fact lists with the first identified as the premise list for the rule while the second provides a consequent list. The TAU cycle includes a step whereby all eligible (primed) IRules are fired. A rule is primed when the members of the premise list are observed to be present in the current fact base. Firing results in the facts in the consequent list being added to, or deleted from, the fact base.

To prevent an elementary kind of infinite loop that would occur if the same primed rule were repeatedly considered for firing, AdaTAU tags an IRule when it has been fired once. A specific IRule tag field is reserved for this purpose. IRules also have a field reserved for a textual explanation of the rule's purpose.

In addition, functions are provided to extract the components of an IRule (*Get_Justification*, *Get_Antecedent*, and *Get_Consequent*). *Set_Flag*, *Clear_Flag*, and *Check_Flag* can be used to manipulate the tag field denoting whether the IRule has fired or not.

QRules

QRules are more complicated because they act in a two stage manner to alter a fact base. Like IRules, a premise fact list is provided that enables a QRule's firing. However, no direct consequent facts are asserted or denied as a consequence of such a firing. Rather, when a QRule fires, a question, whose possible answers are associated with different consequent fact lists, is added to another data structure (the agenda — see below). A reference to the particular question is included in the QRule in place of a consequent fact list.

The actual asking of the question (and resulting processing of facts) is deferred until a later phase in the TAU process. The question data structure is described below, but it should be noted that the response to a question is unknown at the time the rule is written. Thus there can be no one consequent list for a QRule, but rather several. In fact, because the outcome from a QRule hinges on the answer to a question, the consequent lists are actually attached to the question and not the QRule.

Like IRules, QRules are tagged to prevent unlimited refirings. Each QRule supplies a weight factor that contributes to the ranking of the question within the question agenda structure. The weight is used in conjunction with the feature that the same question can be associated with different QRules, each of which can schedule the question using its own weight value to rate the importance of the question. A question's position on the agenda is determined by the sum of the weights assigned to it by all of the QRules requesting it. QRules are also equipped with a single text field that can be used to report

to the user why a particular question was scheduled by this rule.

Functions are also provided to extract the components of a QRule (*Get_Conditions*, *Get_Question*, *Get_Weight*, and *Get_Justification*). *Set_Flag*, *Clear_Flag*, and *Check_Flag*, respectively, mark a QRule as fired, mark a QRule as unfired, and determine whether or not a QRule is marked fired or unfired. Like IRules, a QRule can only be fired once, so marking a QRule fired prevents it from being considered for firing again.

FRules

FRules are unlike IRules and QRules in that they do not cause new facts to be added to a fact base directly or indirectly. Rather, the purpose of an FRule is to promote the opportunity for inference (the basic process of producing new facts from old) in another location within a distributed collection of inference bases. Like QRules, FRules are weighted so that if multiple FRules can fire in a given inference base, the competing focus suggestions can be ranked so that the more likely inference base is examined first. A focus agenda is used to keep track of multiple inference opportunities. If two or more different FRules each point to the same inference base, the sum of the weights of these rules is used to rank the inference context on the focus agenda when each rule is fired. Like both IRules and QRules, FRules are tagged to prevent unlimited refirings and FRules also contain an explanation field.

Operations are provided to extract the components of an FRule (*Get_Conditions*, *Get_Question*, *Get_Weight*, *Get_Context* and *Get_Justification*). *Set_Flag*, *Clear_Flag*, and *Check_Flag*, respectively, mark an FRule as fired, mark an FRule as unfired, and determine whether or not an FRule is marked fired or unfired. Routines are also provided to create FRules from their constituent parts, copy FRules and compare FRules.

Rule Bases

A particular instance of a centralized AdaTAU application is defined by a fact base schema, an optional initial fact base, and a list of rules (IRules, QRules) that are applicable to the fact base. The collections of rules are called rule bases. AdaTAU is configured with two separate rule bases, one for IRules and one for QRules. IRule bases and QRule bases are implemented as instantiations of a generic rule_base package, which in turn instantiates a generic list package.

Distributed AdaTAU applications are defined by collections of inference bases, each of which includes a fact base schema, a fact base, and lists of rules (IRules, QRules and FRules) that are applicable to the fact base. In addition, inference bases also can make use of fact parameters to communicate with each other. FRule bases also are defined via the generic rule base package.

Questions

Questions are directly associated with QRules but are configured separately to allow different rules to use the same question configuration. One component of a question is the text of the question itself which is presented to the user. Another is a structure, called a *response schema*, which maintains, for each possible user's response, a corresponding consequent fact list that describes the consequences of the QRule(s) that scheduled the

asking of the question initially. Response schemas are abstract data types in their own right and are provided, along with appropriate operations, in a separate package.

The simplest kind of question is one that admits a limited number of answers that can be presented in multiple choice terms. For example, a question can be represented as

```
Question Ask_Component_Type is
      Text: {What is the component type?};
      Type: one_of;
      Responses:
              "structure" => (component_type, structure);
              "tool" => (component_type, tool);
              "subsystem" => (component_type, subsystem);
```

For each choice, a corresponding list is stored and members of the list are added/deleted to/from the fact base if the user picks that choice. In the example above, the listed fact is added to the fact base when the user makes the choice to the left of each fact. Facts to be deleted are maked by the '~' character (see the discussion of RBDL in the appendix). An extension of this view permits the user to agree to several (or all) of the possible answer choices. In this case, all of the corresponding consequent lists would be processed, with contradictory facts resulting in a raised exception. Routines are provided to create questions and to extract the various components of a question (*Get_Text*, *Get_Num_Choices*, and *Get_Response_Table*).

Agendas

We have already mentioned that questions are scheduled as a result of the firing of QRules, and that the actual processing of questions is handled via a weighted agenda mechanism. Similarly, FRules produce focus switch suggestions which are handled by a separate focus agenda. The concept of agenda can actually be abstracted as simply a weighted queue of items where the agenda manager will simply provide the most highly weighted item when a user wishes to retrieve an item from the agenda. When an item is added to the agenda, a check is made to see if the item is already on the agenda. If it is, the weight of the item is adjusted. If it is not, the item is added to the agenda along with its initial weight.

In addition, the AdaTAU application requires a link back to the object(s) that posted the item to the agenda. In the initial AdaTAU design, the items on the agenda are questions (actually they are pointers to questions), and the objects posting questions to the agenda are QRules. Thus the question agenda for AdaTAU is an ordered (by weight) list of questions, each of which refers to a list of the QRules that contributed to the question having been placed on the agenda. In the current design, focus agenda items are the identities (usually names) of inference base locations, and a list of FRule reference contexts that identify each of the FRules that caused the agenda item to have been placed on the agenda as well as the inference base location of the FRule itself. The reference lists maintained per agenda item can be used to tag the actual facts that eventually are asserted as a result of using the item retrieved from the agenda as well as permitting truth maintenance under non-monotonic reasoning. In general, knowing the identity of the object that caused the agenda item to be posted or modified can provide information that is necessary when the item is processed later.

The agenda mechanism is provided through a generic package. Agendas are implemented as collections of *agenda records*, which is another private type exported by the agendas package. Each agenda record corresponds to a single agenda item, and vice versa. Agenda records have three components: the agenda item itself, the composite weight associated with the item, and an "information list" (of type *Info_List*, another private type exported by the agendas package). The information list, for question agendas, is a list of the QRules that placed the question on the agenda. This list of QRules also contains the weight that each QRule assigned to the question, but this is not visible to the user.

Many operations are provided both for agenda records and for info_lists. Routines are provided to extract the three components of an agenda record (*Get_Item*, *Get_Weight*, and *Get_Info*). No routines are provided for creating or constructing agenda records because this is done automatically when an item is put on the agenda. Procedures and functions are included to iterate through an Info_List (*Reset_Info*, *More_Info*, and *Next_Info*), to search for information (*Search_Info*), and to test for emptiness (*Empty*).

Create_Agenda is used to create an empty agenda and items are added to the agenda using *Add_Item*. *Delete_Weightiest_Record* removes and returns the item with the highest composite weight from the agenda. The operation *Delete_One_Record* deletes a specified record, not necessarily the first one, from the agenda. The function *Dump_Agenda* converts an agenda to a list of agenda items so that it can be examined using routines from Tau_Lists. *Search_Record* searches for a particular item and, if found, returns the corresponding agenda record. *Empty* is used by the inferencer to determine when the agenda is empty.

A Basic TAU Application Shell

At the outer-most layer, we can outline TAU as an Ada procedure which is invoked with an initial fact base, an IRule base, a QRule base, and a question base. The contents of these bases are determined via translation from RBDL specifications of actual fact base schemas and rule structures for the domain on which TAU is to operate. RBDL is described in the language subsection of this report. A sample Ada procedure implementing a particular method of rule-based inference is as follows. Note that this procedure is a stripped version of the actual Ada version of the Tau procedure to allow the discussion to focus on essential features. For example, parameters that support non-monotonic inference are not shown in the included procedure calls.

```
with Basic_Configuration;

use Basic_Configuration;

-- Basic_Configuration packages the resources required to run a TAU
-- application. These include types for fact bases, both kinds of rule
-- bases and the notions of local question agenda and response management.

procedure Tau (Current_Fb :   in out Fact_Base;
               Irule_Rb :     in     Irule_Base;
               Qrule_Rb :     in     Qrule_Base;
               Questions :    in     Question_Base;
               Local_Agenda : in     Question_Agenda :=
                                     Q_Agendas.Create_Agenda) is

   User_Response : Response;
   Working_Agenda : Question_Agenda := Local_Agenda;

begin

   -- forward chain all firable rules; update agenda

   Think (Current_Fb, Irule_Rb, Qrule_Rb, Local_Agenda);

   while not Q_agendas.Empty (Working_Agenda) loop

   -- ask question at top of agenda producing response

      Ask (Working_Agenda, Questions, User_Response);

   -- use response to modify the fact base

      Update (Current_Fb, User_Response);

   -- Think again

      Think (Current_Fb, Irule_Rb, Qrule_Rb, Working_Agenda);

   end loop;

end Tau;
```

Basic inference progress is made within the Think and Update procedures while user interaction is handled inside of Ask (see figure 2). After an initial execution of the Think phase, a loop through the successive phases of Ask, Update and Think is executed until the working agenda becomes empty. At this point no IRules are primed and no questions derived from QRules remain for the user to answer. We now consider these

phases individually.

Think. An initial invocation of AdaTAU will process all IRules until no further changes to the fact base are possible. IRules will be examined in an arbitrary order; in particular, the rule base designer cannot assume any particular ordering of their being fired. The same arbitrary ordering is followed in all subsequent passes through the list of IRules. During the Think phase, several passes through the set of IRules may be necessary since the addition of facts in the consequent lists of fired IRules may cause other IRules to become primed. Then a single pass over all of the QRules is made so that all of these rules found to be primed can have their associated questions placed on a local agenda that is used to manage an orderly and prioritized interaction with the user. The examination of QRules will also occur in some fixed sequential manner. Multiple passes through this rule set is not required since these rules do not directly affect the fact base. Notice that after the Ask and Update phases are completed, the Think phase is invoked again because the fact base can be changed during the Update phase.

Ask. The user of a TAU-based application must be consulted when no further progress can be made within the Think phase. At this point, the agenda is consulted and a user's response to a question drawn from the agenda is processed. Question-asking and response-recording is handled by the Ask module. Other agenda items, if any, are not processed until after the next Update phase and following Think phase are completed. If

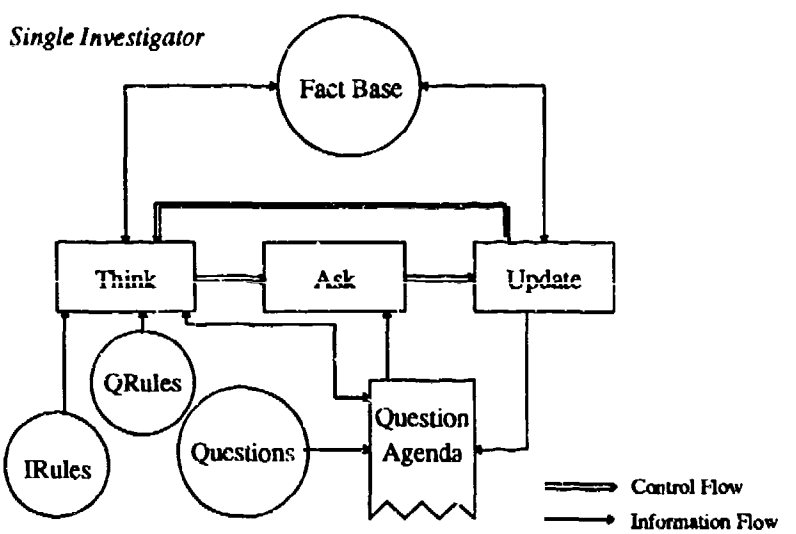


Figure 2. Basic AdaTAU Inference

the agenda is empty initially, and the Think phase does not add any items to the working agenda, the current AdaTAU invocation ends with no further processing.

Update. From the recorded response returned by the Ask module, updates to the current fact base are handled by the procedure Update. Update provides a truth maintenance phase. If a question was asked, depending on the response, consequences traced to the corresponding QRule are processed against the fact base. In the simplest case where no fact deletions occur, the Update phase simply needs to add those consequent facts attached to the particular response obtained from the user. Otherwise, Update must make sure that fact deletions are propagated through a fact dependency table that tracks the origin of facts in the fact base. In any case, the response schema data structure makes the required information easy to obtain.

It will be necessary for the top-level application that requires TAU to provide instances of an initial fact base, accompanying rule bases, and a question base containing all questions which may be posed to the user. Once the TAU component has finished its work (the question agenda becomes empty), the outer application must also process the resulting fact base to extract information to be used subsequently.

An Advanced Application Shell

The advanced TAU organization presented in this section provides capabilities needed by the Reusability Library Framework. The RLF system includes dual knowledge representation schemes in the form of a semantic network providing taxonomic organization for the library domain, and distributed rule base systems that pertain to discrete components of the network. Thus different rule bases will be housed in different portions of the network. In order to prepare for this integrated system, we have designed a version of AdaTAU which employs *Distributed Rule Bases* each of which is processed independently using the AdaTAU model presented previously. Certain facts that are produced during a local inference may need to be transferred and applied elswhere. To accomplish this, a fact parameter capability is introduced through which facts may be exported from an inference base, imported to an inference base, or both.

A separate process called the focuser coordinates the results produced by processing the individual rule bases (see figure 3). The focuser is guided by an agenda of (the identities of the) separate inference base components. An item on the focuser's agenda is able to direct the focuser to the inference base that is most likely to permit further inference progress. Like the question agenda used within basic AdaTAU, items on the focuser's agenda are also weighted so that the inference base with the most potential to affect the fact base is consulted next. Individual rule base processing is able to affect the focus agenda through the firing of FRules which generate focus switch suggestions that are merged in with the current focus agenda.

Although not indicated in the diagram, the occurrence of a context switch includes the exportation of facts from the current fact base and the importation of facts to the new fact base. This transfer is accomplished through the fact parameter facility provided in Distributed AdaTAU. The rule base invocation strategy is designed to permit the application to start/suspend/resume separate TAU interactions using the individual rule base components. A key feature of this strategy is that these local TAU interactions will not be "greedy"; that is, exhausting all possibilities locally before considering TAU

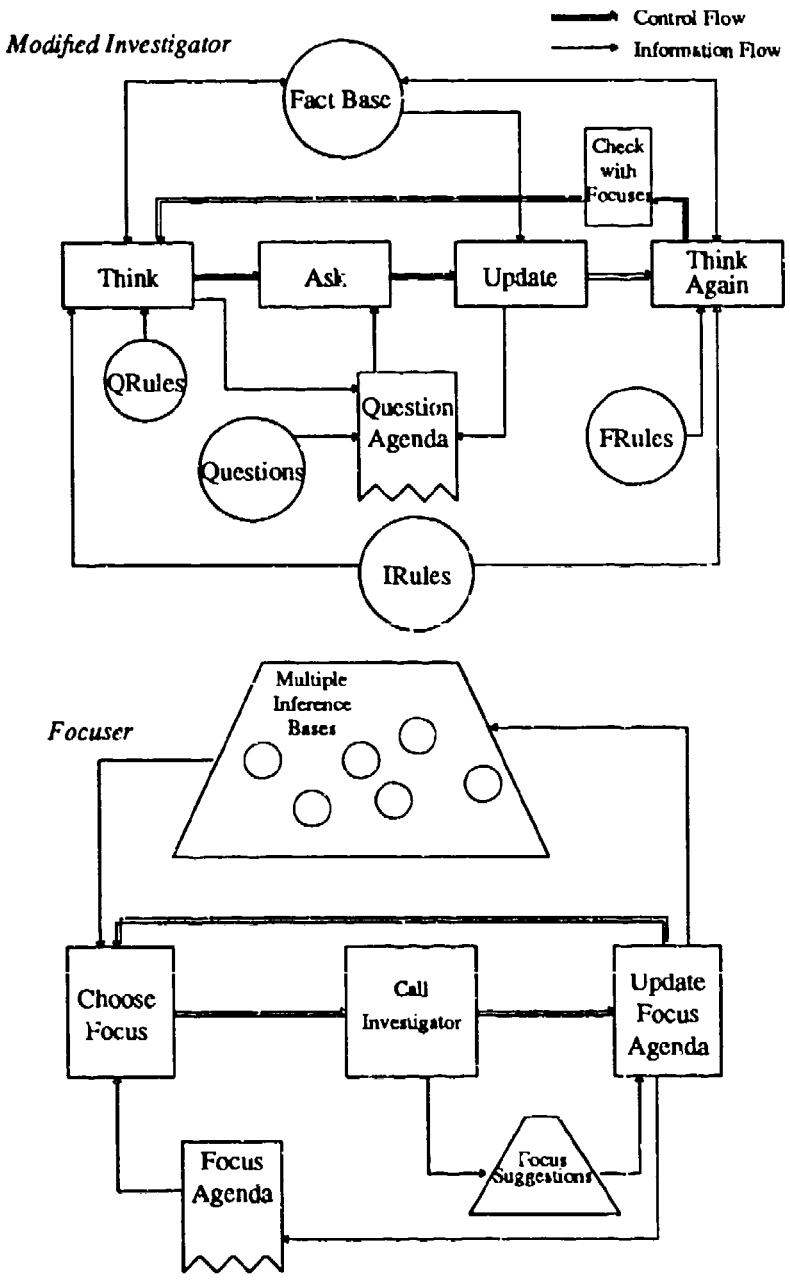


Figure 3. Distributed AdaTAU Inference

components elsewhere in the system. Instead, the system will operate on a "willing surrender" strategy that permits the controlling application to expect context switches after a single pass through the Think, Ask, Update sequence. The means to providing this capability is the focus suggestion agenda.

We assume that any application using the distributed form of AdaTAU (abbreviated DTAU) will deposit the identity of an initial inference base on the focus agenda so that an initial execution of the TAU sequence can occur relative to this start-up inference base. The version of DTAU presented here is a skeletal representation that integrates the processing required for distributed rule bases with the local investigation that occurs once a particular rule base is selected. The actual code for DTAU is considerably more complicated. This version merges the modified investigator shown in Figure 3 with the focuser into a single module and ignores details pertaining to non-monotonic inferencing.

```
with basic_configuration,
     advanced_configuration_instance;

use basic_configuration,
    advanced_configuration_instance;

-- advanced_configuration packages the resources required
-- to run a distributed TAU application including focus management.
-- These include types for fact bases, all three kinds of rule bases,
-- and the notions of local question agenda,
-- global (re-focus) agenda, and response management.

procedure DTAU (Current_Fb : in out Basic_Configuration.Tau_Fact_Base;
                Global_Agenda          : in out App_Focus_Agenda;
                Current_Inference_Base : in out App_Inference_Base) is

   user_response : response;
   first_choice, second_choice :
                          inference_base :=
                                get_inference_base (global_agenda);
   irule_RB : irule_base;
   qrule_RB : qrule_base;
   frule_RB : frule_base;
   questions : question_base;
   local_agenda : question_agenda;
   refocus_suggestions : focus_suggestions :=
                                NULL_FOCUS_SUGGESTION;

begin

   -- attempt to establish an initial TAU identity

   if Current_Inference_Base = NULL_INFERENCE_BASE then
```

```
      -- There must be an initial inference base to get started

      return;

   else

      -- Translate TAU identity fetched from global agenda to a TAU context

      Acquire_TAU_Context (Current_Inference_Base, irule_RB, qrule_RB,
                           frule_RB, questions, local_agenda);

      Think (current_FB, irule_RB, qrule_RB, local_agenda);

      loop

         -- This is the basic investigator sequence: Think, Ask, Update,
         -- Think again

         -- forward chain all firable rules; update local agenda;
         -- QRules are considered in the Think module,
              Rules are considered in the Think_After module.
              re IRules may have become primed after user is consulted.


         if not q_agendas.empty (local_agenda) then

          -- ask question at top of agenda producing response

             Ask (local_agenda, questions, user_response);

          -- use response to modify the fact base

             Update (current_FB, user_response);

         end if;

          -- Now process all newly fireable irules and fire
          -- any primed frules. Any fired frules
          -- will contribute re-focus suggestions

         Think_After (current_FB, irule_RB, frule_RB, refocus_suggestions);

         -- Merge refocus suggestions with current global agenda to
         -- reconsider most likely investigator bases;
         -- Return handles to first two choices

         eval_focus (global_agenda, refocus_suggestions, first_choice,
```

```
                                                    second_choice);

        if not (first_choice = NULL_INFERENCE_BASE) then

            -- Perform context switch since a refocus is indicated

            Update_TAU_Context (current_inference_base, irule_RB, qrule_RB,
                   frule_RB, questions, local_agenda);

            current_inference_base := first_choice;

            Acquire_TAU_Context (current_inference_base, irule_RB, qrule_RB,
                   frule_RB, questions, local_agenda);

        end if;

        Think (current_FB, irule_RB, qrule_RB, local_agenda);

        -- Think again before leaving or continuing the loop
        -- Stop when no further inference processing can take place

        exit when (first_choice = NULL_INFERENCE_BASE) and
                  basic_configuration.q_agendas.empty (local_agenda);

        -- Now test the three failed exiting conditions carefully

    end loop;

  end if;

end DTAU;
```

The Think, Ask and Update modules for this advanced AdaTAU interaction scheme function just as they did for the basic centralized version of AdaTAU. The Think_After module functions almost identically to Think except that after all IRules are considered as described for Think, FRules are processed which generate new, or additional, focus switch suggestions that are to be merged with the current focus agenda. The main part of DTAU is structured as an Ada loop with explicit exit. When the indicated conditions occur, no further inference progress can be made within the fact base.

The actual consideration and management of focus switches is assigned to the procedure eval_focus. Eval_focus returns with the top choice for which inference base should be considered next. Typically, where there are only a few inference base possibilities, it is likely that the top choice will be the same as the current inference base, and the local agenda will have items remaining on it. In this case, inference will continue by executing the basic TAU sequence again. When a context switch is required, the old context information is saved as necessary (by Update_TAU_Context) and a new context established by calling Acquire_TAU_Context. The design currently relies on

Acquire_TAU_Context to produce the actual rule bases and question bases that are required by the individual TAU modules.

4. AdaTAU System Components

The major conceptual components of AdaTAU are mapped onto compatible Ada definitions that take advantage of relevant Ada features. For the most part, AdaTAU concepts are effectively captured by Ada packages that provide an abstract data type view of the concept. At the core of AdaTAU, we provide an integrable family of Ada packages, each of which is identified with a key concept of the AdaTAU approach.

Identifying the data objects to be manipulated by the AdaTAU inferencer is a very straightforward task. The objects correspond directly to the activities that take place during an inference cycle. These activities involve firing rules, taking questions from a local question agenda and asking them, processing inference base context switches through the focus agenda and processing facts within a fact base. Consequently, the objects that the AdaTAU inferencer is most directly concerned with are facts and fact bases, rules (IRules, QRules and FRules), rule bases, and agendas. The corresponding ADTs, however, build upon and are associated with other objects as well as each other (see figure 4). Note that this figure only depicts the basic AdaTAU configuration and does not show packages that make up the advanced AdaTAU configuration.

For example, facts and fact base schemas are the components which constitute fact bases. Facts also make up fact lists, which in turn are components of both IRules and QRules. Response schemas are a component of questions, which are in turn an additional component of QRules. IRules and QRules are used to instantiate their respective types of rule bases and rule lists. Rules of both types are associated with fact bases when they are fired. Rule bases are converted to rule lists when they need to be examined by higher-level routines. Fact bases can be converted to fact lists in a similar manner.

The routines included in these ADT packages are designed to provide the basic minimal primitive operations needed by any application utilizing the ADTs. The provided operations fall into two general categories: those that modify an object (creating, adding, deleting, etc.) and those used to examine the objects (extracting components, iterating through lists and tables, etc.). In the AdaTAU application, the modifying operations will for the most part be used by the RBDL processor to create and build objects and by higher-level inferencer routines to update fact bases. The routines for examining objects, on the other hand, will be used by the inferencer and for debugging by AdaTAU builders.

Each ADT package contains routines for converting objects to a structure suitable for storing in a file and for manipulating tables of such persistent structures. Using these routines, the application writer can initialize these tables, add, delete, and retrieve objects from tables, load a table from a file, store a table to a file, and delete a table (thus freeing the memory it occupies). This scheme allows the state of an AdaTAU session to be frozen and stored, and then used to start up the session at a later time.

There are some characteristics that are common to all of these ADTs. One is that the standard "=" and ":=" operations use "share" semantics. That is, two objects are not equal ("=") unless they are actually the *same* object. Similarly, assignment (":=") results in the left-hand side of the assignment simply referring to the same object as that represented by the right-hand side. Operations which provide "copy" semantics are provided in all packages. These operations are *equivalent* (which returns true if two objects are identical, but not necessarily the same object) and *copy* (which returns an identical,

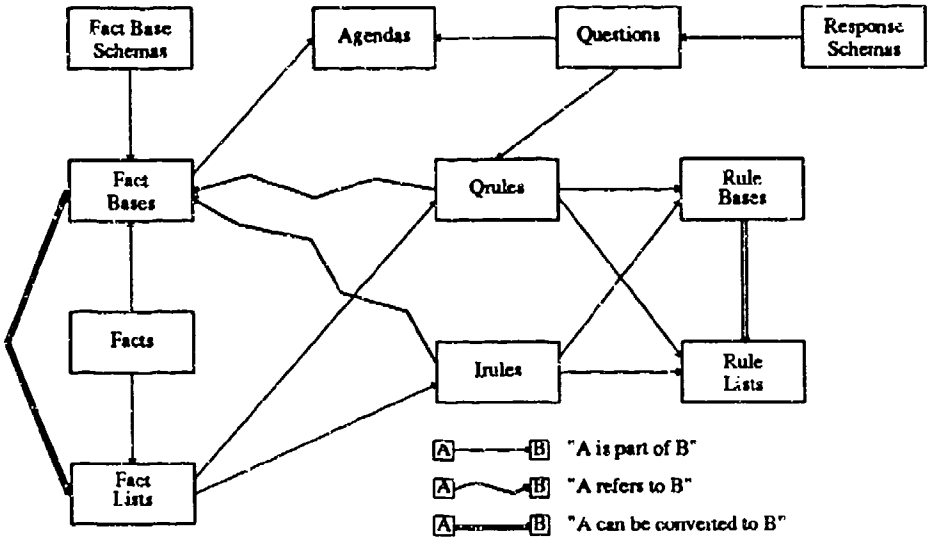


Figure 4. Basic AdaTAU Component Taxonomy

but separate, copy of its input parameter).

All ADTs also have an associated *create* operation which performs any implementation-dependent initialization necessary. Each object must be "created", using this routine, before it can be used in any other operation. After an object is declared, but before it is "created", it is said to be null. Each ADT package provides a constant which represents this null value, so an object can be compared to this null constant (using "=") to determine whether or not it has been created. Where appropriate, an exception is raised if an operation is attempted on a null object.

All of the operations that add to or delete from an object are "hard". That is, an "add" operation raises an exception if the component to be added already exists within the object (unless it makes sense for a particular object to have multiple copies of identical components, as is the case in a few situations). Similarly, a "delete" operation raises an exception if the component to be deleted does not exist within the object. "Probe" functions are also provided which test for the conditions which would lead to an exception.

Each ADT is described in more detail in the separate subsections which follow. For each ADT (Ada package), the following information is provided:

- the package name of the ADT

- a brief description of the ADT
- the major objects and types of the ADT
- the major operations supported by the ADT.

Each of the objects and operations is summarized with a brief explanation clause. More information about these objects and operations can be found within the Ada source files containing the actual package specifications. The information presented in this document is minimal compared to the description given in the ADT itself, and corresponds to a snapshot of a particular version of the ADTs. Information presented here may be incomplete, or there may be inconsistencies between this document and the latest versions of the ADTs themselves. As such, the interested reader is encouraged to read the ADTs for the most up-to-date and complete information about each ADT. Each file is formatted with a standard header and includes comprehensive descriptions of the contents and basic semantics of each package.

Several of the ADTs require that collections of objects of a particular type must be maintained; the *Tau_Lists* generic package has been provided to implement all the list structures used in AdaTAU. The data structure provided is a simple, singly-linked list. Operations are provided for creating empty Tau_Lists (*Create_Tau_List*), adding items to the list (*Build_List*), iterating through a list (*Reset*, *More_Items*, and *Next_Item*), searching for a particular item (*Search_Item*), deleting a particular item (*Delete_Item*), and testing a list for emptiness (*Empty*).

4.1. Facts

Package Name

Facts

Description

This package defines the abstract data type fact. Facts, conceptually, are made up of two parts: an attribute and a value. For example, the pair (loop_type, while) can be viewed as a fact stating that the loop under consideration is a while loop. Routines are provided for creating facts, as well as for extracting the attributes and values of facts. Functions are also provided to convert attributes and values to strings and vice versa. Additionally, a routine is provided to extract a fact's unique_identifier value. Finally, persistence routines are included that are used by higher-level packages to save files of facts.

Objects

```
type Attr_Type is private;
      --This is the type of the attribute part of a fact.

type Value_Type is private;
      --This is the type of the value part of a fact.

type Fact is private;
      --A fact can be thought of as an (attribute, value) pair.

type Fact_Table is private;
      --Facts are saved to files in the form of fact tables

Null_Attr : constant Attr_Type;

Null_Value : constant Value_Type;

Null_Fact : constant Fact;
      --This is the value of a fact before it is "created"
```

Operations

`Convert_Attr` (2)
: two overloaded operations under this name; one converts strings to objects of type attr_type, the other converts objects of type attr_type to strings.

`Convert_Value` (2)
: two overloaded operations under this name; one converts strings to objects of type value_type, the other converts objects of type value_type to strings.

`Create`
: creates a fact from a given attribute and value.

`Copy`
: returns an identical, but separate, copy of a fact.

`Get_Id`
: returns the unique_identifier component of a fact.

`Get_Attr`
: returns the attribute part of a fact.

`Get_Value`
returns the value part of a fact.

`Equivalent`
returns true if the attributes and values of two facts are identical.

`Create_Fact_Table`
creates an empty fact table.

`Save_Fact_Table`
saves table in a file identified by a name string.

`Open_Table`
retrieves table from a file identified by a name string and prepares it for reading.

`Delete_Table`
releases the memory used for table.

`Store_Fact`
stores a fact in table.

`Retrieve_By_Id`
returns a fact with a specific identifier id which is in a table. Returns NULL_FACT if no such fact exists.

`More_Entries`
returns true if there are more facts to be returned from table.

`Next_Entry`
retrieves, through a specified parameter next, the next fact in table.

4.2. Fact Lists

Package Name

Fact_Lists

Description

This package defines the abstract data type fact_list. A Fact_List is simply a list of facts to be used where such lists are needed in rules, etc., (e.g., antecedents, consequents) and to provide a way to examine the facts in a fact base. They are also used various other places in the AdaTAU system. The implementation of fact lists relies on an instantiation of the generic package Tau_Lists, instantiated with the type fact. Routines for creating and building lists, as well as for getting the first fact in the list, for getting each successive fact in the list, for searching for a particular fact, for deleting a fact from the list, and for determining if a list is empty or not, are all provided through the Tau_Lists package. A function for finding the fact associated with a particular attribute in the list is provided through this package.

Objects

```
package New_Lists is new Tau_Lists (Facts.Fact, Facts.Equivalent);
        -- Tau_Lists is instantiated to implement Fact_Lists

subtype Fact_List is New_Lists.Tau_List;
        -- the actual fact list type
```

Operations

Copy
: makes an identical, but separate, copy of a fact list.

Null_And_Free
: produces a null list and releases any dynamic storage associated with the list.

Merge_Lists
: merges two fact lists together into a single list.

Equivalent
: determines whether or not two fact lists contain equivalent facts in the same order.

Same_Facts
: determines whether or not two fact lists contain equivalent facts, without regard to order.

Search_Value
: given a list and an attribute, finds the value(s) associated with that attribute in the list.

4.3. Fact Value Lists

Package Name

Fact_Value_Lists

Description

Instantiates Tau_Lists with fact values. This allows lists of fact values to be created independent of a given fact schema (for example, in the fact parameters package).

Objects

```
package Fact_Value_Lists_Pkg is new Tau_Lists (Facts.Value_Type);
   -- Equivalence operation defaults to "=", with copy semantics)

subtype Value_List is Fact_Value_Lists_Pkg.Tau_List;

Null_Value_List      : constant Value_List := Fact_Value_Lists_Pkg.Null_List;
```

Operations

Copy_One_Val
: copies a single fact value.

others
: derived from the Fact_Value_Lists_Pkg instantiation of the generic package Tau_Lists.

4.4. Fact Schemas

Package Name

Fact_Schemas

Description

This package defines the abstract data type Fact_Schema. A Fact_Schema is a structure which defines the form of any fact(s) associated with a particular attribute. A Fact_Schema includes the name of the attribute, an indication of whether it is multi-valued or single-valued or can take on any arbitrary value, and an optional list of values that the fact can take on. Routines are provided to create a fact schema and to retrieve all of the various components of a fact schema. As well, a function is included which checks a fact for validity with respect to the fact schema. Persistence routines are also included that are used by higher-level packages to save files of Fact_Schemas.

Objects

```
package Value_Lists renames Fact_Value_Lists.Fact_Value_Lists_Pkg;

subtype Value_List is Fact_Value_Lists.Value_List;
      --This is the type of the list of values that an attribute
      --can take on.

type Fact_Type is (One_Of, Some_Of, Any);
      --This indicates whether the fact can take on one of several
      --user-enumerated values, some of several user-enumerated
      --values, or any arbitrary value.

type Fact_Schema is private;
      --This type holds the schema associated with a single attribute.

type Schema_Table is private;
      --Fact_schemas are saved to files in the form of schema tables

Null_Fact_Schema  : constant Fact_Schema;
```

Operations

Create
: creates a fact schema for a given attribute, with a given fact_type and, in some cases, a list of possible values.

Null_And_Free
: nullifies a fact schema and releases any dynamic storage associated with it.

Copy
: creates an identical, but separate, copy of its input parameter.

Get_Attr
: extracts the attribute from a fact schema.

Get_Type
: extracts the fact_type from a fact schema.

`Get_Values`
extracts the list of values from a fact schema.

`Equivalent`
compares the attributes, fact_types, and value lists of two fact schemas to see if they are equivalent.

`Valid`
checks a fact for validity with respect to schema.

`Create_Schema_Table`
creates an empty schema table.

`Save_Schema_Table`
saves a schema table in files.

`Open_Table`
retrieves a schema table and prepares it for reading.

`Delete_Table`
releases the memory used for a schema table.

`Store_Schema`
stores a fact_schema in a schema table.

`More_Entries`
determines if there are more fact_schemas to be returned using next_entry.

`Next_Entry`
retrieves the next fact_schema in a schema table.

4.5. Fact Base Schemas

Package Name

Fact_Base_Schemas

Description

This package defines the abstract data type Fact_Base_Schema. A Fact_Base_Schema is a component of a fact base and dictates what facts are allowed to be added to the fact base. It defines which facts are single-valued and which are multi-valued, which must take values from a set of values enumerated by the user and which may take any arbitrary value. It also defines what attribute values are allowable. A Fact_Base_Schema is a collection of Fact_Schemas (from the package Fact_Schemas), each of which defines the schema associated with a particular attribute. As well, a Fact_Base_Schema is a component of a Fact_Base, defined in the package Fact_Bases, and defines the form of facts that can be placed in the fact base. Routines are provided to create an empty fact base schema, to add a fact schema to a fact base schema, to convert a fact base schema to a list of fact schemas (to facilitate examining the schema), to search for a fact schema associated with a particular attribute, and to check a fact base schema for emptiness. There is also a group of routines included which check facts, fact schemas, attributes, and fact lists for validity with respect to a fact base schema.

Objects

```
type Fact_Base_Schema is private;
    --This structure holds the schema for an entire fact base.

package Schema_Lists is new Tau_Lists (Fact_Schemas.Fact_Schema,
                                       Fact_Schemas.Equivalent);

subtype Schema_List is Schema_Lists.Tau_List;
   --List form of collection of individual schemas

Null_Fact_Base_Schema : constant Fact_Base_Schema;
```

Operations

`Create_Fact_Base_Schema`
creates an empty fact base schema.

`Compose_Fact_Base_Schema`
composes fact base schema, according to the schema list provided, which can be further filled with repeated calls to add_schema.

`Copy`
creates an identical, but separate, copy of its input parameter.

`Null_And_Free`
nullify, and release associated dynamic memory, for a particular fact base schema.

`Add_Schema`
adds a fact schema to a fact base schema.

`Change_Label`
change the label of a fact base schema.

`Dump_Schema`
converts a fact base schema to a list of fact schemas in order that the fact base schema can be more easily examined.

`Get_Label`
returns a fact base schema's label.

`Get_Id`
returns the fact base schema's unique identifier.

`Get_Schema`
returns the fact_schema associated with the attribute in the fact base schema parameter.

`Empty`
checks a fact base schema for emptiness.

`Valid` (4)
four different operations under this overloaded name; checks a fact, attribute, fact list, or fact schema for validity with respect to a fact base schema.

`Equivalent`
checks all of the components of two fact base schemas to see if they are equivalent.

4.6. Fact Bases

Package Name

Fact_Bases

Description

This package defines the abstract data type Fact_Base. A fact base is simply a collection of facts with an associated fact base schema which is used to monitor the validity, or admissability of facts that are added to the fact base. Fact bases also have a label and an internal identifier. Fact bases can be used to collect facts into logically related groups. A routine is provided for creating fact bases. A fact base can be built by successively putting facts into it using the install routines. A function is provided to check if a fact is valid for a particular fact base. Routines are also provided for searching for a particular fact, finding a value for a particular attribute, deleting a fact from the fact base, extracting the fact base schema, label, and identifier from a fact base, changing a label, and determining if a fact base is empty or not. If the individual facts in the fact base need to be examined, then the base can be converted to a Fact_List (from the package Fact_Lists), using Dump_Base, and then examined using list routines. Operations are also provided to determine if two fact bases have equivalent fact base schemas and consistent facts. Set operations are also provided for fact bases. Some operations are also included which are used by higher level packages to save persistent versions of fact bases to files.

Objects

```
type Fact_Base is private;
      --A fact_base can be thought of simply as an unordered
      --collection of facts, along with a schema defining what facts
      --are admissable.

subtype Base_List is Fact_Lists.Fact_List;
      --just the facts in a fact base

Null_Fact_Base : constant Fact_Base;

Null_Base_List    : constant Base_List :=
                                  Fact_Lists.New_Lists.Null_List;
```

Operations

`Create`
: creates an empty fact base with the given fact base schema.

`Compose`
: composes a fact base, according to the constituents provided, which can be further filled with repeated calls to install.

`Install` (2)
: this name is overloaded with two operations; one adds the a parameter of type fact to a fact base; the other adds an attribute - value pair to a fact base.

`Union`
: creates the union of two fact bases.

`Intersection`
creates the intersection of two fact bases.

`Difference`
creates the difference of two fact bases.

`Copy`
creates an identical, but separate, copy of its input parameter.

`Null_And_Free`
nullifies a fact base releasing any storage occupied by the fact base.

`Delete_Fact`
deletes a fact from the fact base.

`Change_Label`
change the label of a fact base.

`Dump_Base`
converts a fact base to a fact list for easier examination of the facts.

`Get_Label`
returns the label of a given fact base.

`Get_Id`
returns the fact base's unique identifier.

`Get_Schema`
returns the fact base schema of a given fact base.

`Search_Fact` (2)
this name is overloaded with two operations; one searches for a given fact in the fact base; the other searches for a given attribute.

`Search_Facts`
returns all the facts associated with a given attribute.

`Subset`
determines if one given fact base is a subset of another.

`Empty`
checks a fact base for emptiness.

`Valid`
checks a given fact for validity with respect to a given fact base.

`Consistent`
determines if two fact bases have consistent facts.

`Compatible`
determines if two fact bases have equvalent fact base schemas.

`Equivalent`
checks the fact base schemas and all the facts associated with two fact bases to determine if they are equivalent.

4.7. Fact Parameters

Package Name

Fact_Parameters

Description

Fact parameters are essentially requests for facts required by or provided by an AdaTAU inferencer. Each fact parameter specifies the attribute for which a corresponding value is required at the time the parameter is imported or exported. The "protocol" of the fact parameter indicates the desired reaction to the absence of a value for the desired attribute: silent failure (Optional Protocol), provision of a default (Default Protocol), or raising an exception if the fact parameter is explicitly required for the inference to be valid (Mandatory Protocol). The defaults are specified via fact_value_lists, which serve for "one_of", "some_of" and "any" type facts.

Objects

```
type Fact_Parameter_Protocol is (
      Mandatory,   -- absence of corresponding fact raises exception
      Default,     -- absence result in default value(s)
      Optional     -- absence results in no action
      );

type Fact_Parameter(Protocol : Fact_Parameter_Protocol := Mandatory )
       is private;
        -- the type itself

Null_Fact_Parameter : constant Fact_Parameter;
```

Operations

`Create` (4)
: overloaded for various types of integrity checking; one ignores checking any schemas, one checks only a single fact schema, one checks a single fact base schema, one checks two fact base schemas in order to allow a legal exchange between two different fact base schemas.

`Attribute`
: returns the fact attribute of the parameter.

`Protocol`
: returns the fact protocol of the parameter.

`Default_Values`
: returns the default value of the fact parameter.

`Search_Fact` (2)
: this name is overloaded with two operations; one returns a fact corresponding to a fact parameter in a fact base, over-riding the protocol specified in Parameter with Protocol; the other returns a fact corresponding to a fact parameter in a fact base, according to Fact_Protocol of the fact parameter.

`Valid` (2)
: this name is overloaded with two operations; one reports whether the parameter that would be formed from the specified attribute, protocol, and default value would be valid with respect to the given fact base schema; the other reports whether the parameter is valid for the fact base schema.

`Created`
Checks whether a successful Create was applied to the fact parameter.

`Transfer`
moves facts between fact bases via fact parameters.

4.8. Fact Parameter Lists

Package Name

Fact_Parameter_Lists

Description

Provides a composite layer of operations over fact_parameters, for operating on lists of fact parameters.

Objects

```
package New_Lists is new Tau_Lists (Fact_Parameters.Fact_Parameter);
      -- Equivalence operation defaults to "=", with copy semantics

subtype Fact_Parameter_List is New_Lists.Tau_List;
        -- the type itself
```

Operations

none except as provided in the Tau_Lists abstraction.

4.9. IRules

Package Name

Irules

Description

This package defines the abstract data type Irule, or inference rule. Irules, conceptually, can be thought of as two sets of facts: the antecedent (the facts that must be in the fact base in order for the rule to fire) and the consequent (the facts that are inferred and added to the fact base when the rule fires). Irules also have an external label which is strictly for the convenience of the user and is not used internally at all. There also is an internal unique identifier for each Irule. Irules also contain an internal flag which signals whether or not the rule has been fired. Another component of an Irule is a textual justification. Irules can be created using the routine create. Before this, however, two Fact_Lists (from the package Fact_Lists) must be built and then passed to the create routine as the rule's antecedent and consequent. Routines are also provided to extract the label, the identifier, the antecedent, the consequent and the justification from an Irule, and to set, clear, and check the fired flag and to change the label. Persistence routines are also included that are used by higher-level packages to save files of Irules.

Objects

```
subtype Text is String (1..400);
      --used for an Irule justification

type Irule is private;
      --An Irule can be thought of as two sets of facts, an antecedent
      --and a consequent.

type Irule_Table is private;
      --Irules are saved to files in the form of Irule tables

Null_Irule : constant Irule;
```

Operations

`Create`
creates an Irule from two fact lists and a label.

`Copy`
creates an identical, but separate, copy of its input parameter.

`Null_And_Free`
replaces an irule with a null irule, freeing any dynamic storage occupied by irule contents.

`Change_Label`
change an Irule label.

`Set_Flag`
sets the "fired" flag of an Irule to true.

`Clear_Flag`
sets the "fired" flag of an Irule to false.

`Get_Label`
extract the label from an Irule.

`Get_Id`
extract the id from an Irule.

`Get_Justification`
extract the justification from an Irule.

`Get_Antecedent`
extract the antecedent from an Irule.

`Get_Consequent`
extract the consequent from an Irule.

`Check_Flag`
checks the "fired" flag of an Irule.

`Equivalent`
checks the antecedents and the consequents of two Irules to determine if they are equivalent.

`Create_Irule_Table`
creates an empty Irule table.

`Save_Irule_Table`
saves an Irule table in files.

`Open_Table`
retrieves an Irule table and prepares it for reading.

`Delete_Table`
releases the memory used for an Irule table.

`Store_Irule`
stores an Irule in an Irule table.

`More_Entries`
determines if there are more Irules to be returned using next_entry.

`Next_Entry`
retrieves the next Irule in an Irule table.

4.10. FRules

Package Name

Frules

Description

This package defines the abstract data type Frule, or focus_suggesting rule. Frules, conceptually, can be thought of as having several components: a set of conditions, or facts, that must be in the fact base in order for the rule to fire, an inference base identity referencing another TAU inference context wherein inferencing can either be begun or continued (captured as another ADT implemented as the package inference_bases), a numerical weight, and an English explanation of why this context switch should be considered in this particular instance. This inference base identity is supplied as a generic parameter. Frules also contain an internal flag which signals whether or not the rule has been fired. Routines are provided to create Frules and to extract the different components that are visible to the user. Routines are also provided to set, clear, and check the "fired" flag in a Frule.

Objects

```
type Text is new String (1 .. 400);
     --Used anywhere text is needed.

type Frule is private;
     --A frule can be thought of as a set of facts (an antecedent),
     --an inference base suggestion, a numerical weight, and a justification
     --in English.

type Frule_Context is private;
     --An frule context associates an frule and an inference context

subtype Frule_Site_Context is Frule_Context;
     --An frule site context joins together an frule and an inference
     --context that it was applied in

subtype Frule_Switch_Context is Frule_Context;
     --An frule switch context joins together an frule and an inference
     --context that it suggests

type Frule_Table is private;
     --Frules are saved to files in the form of Frule tables

Null_Text          : constant Text := Text'(1 .. 3 => '?',
                                           others => ' ');

Null_Frule         : constant Frule;
```

Operations

Create

creates a new frule with the given components.

`Create_Site_Context`
creates a new frule_site_context with the given components.

`Create_Switch_Context`
creates a new frule_switch_context from the given components.

`Copy`
returns an frule which is identical to, but separate from, the rule parameters.

`Copy_Frule_Switch_Context`
returns a frule_context which is identical to, but separate from it's input.

`Null_And_Free`
replaces Rule with a Null_Frule, freeing any storage associated with the rule

`Change_Label`
replaces the label of rule with new_label.

`Set_Flag`
sets the flag signalling whether or not the rule has fired to true.

`Clear_Flag`
sets the flag signalling whether or not the rule has fired to false.

`Get_Label`
extracts the label of a frule.

`Get_Id`
extracts the identifier of a frule.

`Get_Conditions`
extracts the fact list that is the set of conditions needed to be satisfied for the rule to fire.

`Get_Context`
extracts the inference_base identity to be suggested when the rule fires (this type is the generic parameter to this procedure).

`Get_Weight`
extracts the weight, or priority, of the frule.

`Get_Justification`
extracts the justification associated with the frule.

`Check_Flag`
returns the value of the flag signalling whether or not the rule has fired.

`Get_Switch_Context_Id`
returns the context id of the switch context.

`Get_Switch_Context_Frule`
returns the frule id of the frule which suggested the context switch recorded in the switch context.

`Get_Site_Context_Id`
returns the context id of the site context.

`Get_Site_Context_Frule`
returns the frule id of the frule associated with the site in the site context.

`Equivalent`
returns true if the conditions, inference base contexts, and weights of the two rule parameters are equal.

`Equal_Frule_Switch_Context`
returns true if the two frule_context's are equal.

`Create_Frule_Table`
creates an empty frule table.

`Save_Frule_Table`
saves table in several files, using name to generate the file names.

`Open_Table`
: using file name parameter to generate the file names, retrieves table from several files and prepares it for reading.

`Delete_Table`
: releases the memory used for table.

`Store_Frule`
: stores rule in table.

`More_Entries`
: returns true if there are more frules to be returned from table.

`Next_Entry`
: retrieves, in parameter next, the next frule in table.

4.11. QRules

Package Name

Qrules

Description

This package defines the abstract data type Qrule, or question-asking rule. Qrules, conceptually, can be thought of as having several components: a set of conditions, or facts, that must be in the fact base in order for the rule to fire, a question to be asked (a question is another ADT, described in the package questions), a numerical weight, and an English explanation of why this question should be asked in this particular instance. As well, the Qrule contains an external string identifier which is strictly for the convenience of the user and is not used internally at all. There also is an internal unique identifier for each Qrule. Qrules also contain an internal flag which signals whether or not the rule has been fired. Routines are provided to create Qrules and to extract the different components that are visible to the user. Routines are also provided to set, clear, and check the "fired" flag in a Qrule, as well as to change a Qrule's label. Persistence routines are also included that are used by higher-level packages to save files of Qrules.

Objects

```
subtype Text is String(1..400);
      --Used for Qrule justification.

type Qrule is private;
      --A Qrule can be thought of as a set of facts (an antecedent),
      --a question to be asked, a numerical weight, and a justification
      --in English.

type Q ule_Table is private;
      --Qrules are saved to files in the form of Qrule tables

Null_Text : constant Text := Text'(1 .. 3 => '?', others => ' ');
      --An undefined text value

Null_Qrule : constant Qrule;
```

Operations

`Create`
creates a Qrule from its constituents.

`Copy`
creates an identical, but separate, copy of its input parameter.

`Null_And_Free`
replaces a qrule with a null qrule, freeing any dynamic storage occupied by qrule contents.

`Change_Label`
change a Qrule label.

`Set_Flag`
sets the "fired" flag of a Qrule to true.

`Clear_Flag`
sets the "fired" flag of a Qrule to false.

`Get_Label`
extract the label from a Qrule.

`Get_Id`
extract the id from a Qrule.

`Get_Conditions`
extracts the list of condition facts from a Qrule.

`Get_Question`
extracts the question from a Qrule.

`Get_Weight`
extracts the weight from a Qrule.

`Get_Justification`
extract the justification from a Qrule.

`Check_Flag`
checks the "fired" flag of a Qrule.

`Equivalent`
checks the antecedents and the consequents of two Qrules to determine if they are equivalent.

`Create_Qrule_Table`
creates an empty Qrule table.

`Save_Qrule_Table`
saves a Qrule table in files.

`Open_Table`
retrieves a Qrule table and prepares it for reading.

`Delete_Table`
releases the memory used for a Qrule table.

`Store_Qrule`
stores a Qrule in a Qrule table.

`More_Entries`
determines if there are more Qrules to be returned using next_entry.

`Next_Entry`
retrieves the next Qrule in a Qrule table.

4.12. Questions

Package Name

Questions

Description

This package defines the abstract data type Question. Questions are comprised of the text of the question and a response table, which contains the possible answers to the question and a list of facts to be asserted for each answer. The response table is of type Response_Schema, defined in the package Response_Schemas. Each question also contains an indication of whether the user may choose just one or more than one answer to the question. Additionally, questions have an external string identifier which is strictly for the convenience of the user and is not used internally at all. There is also is an internal unique identifier for each question which is generated when the question is created. Routines are provided to create questions, to extract the various components from a question, and to change the label of a question. Persistence routines are also included that are used by higher-level packages to save files of questions.

Objects

```
type Text is new String (1 .. 400);
      --Used for question text.

type Num_Choices is (One_Of, Some_Of);
      --This type indicates whether the question is a "one_of" or a
      --"some_of" question, i.e., whether the user can pick just one
      --or more than one answer.

type Question is private;
      --A question includes the English text of the question and a
      --response table, representing the possible answers and the
      --facts to be asserted for each answer.

type Question_Table is private;
      --Questions are saved to files in the form of question tables

Null_Text : constant Text := Text'(1 .. 3 => '?',
                                   others => ' ');
      --Value used for undefined text

Null_Question     : constant Question;
```

Operations

`Create`
creates a question from the given label, text, number of choices, and response schema.

`Copy`
creates an identical, but separate, copy of its input parameter.

`Null_And_Free`
replaces a question with a null question, freeing any dynamic storage occupied by question contents.

`Change_Label`
change a question label.

`Get_Label`
extract the label from a question.

`Get_Id`
extract the id from an question.

`Get_Text`
extract the text from a question.

`Get_Num_Choices`
determines whether a given question is "one of" or "some of".

`Get_Response_Table`
extracts the response schema from a question.

`Equivalent`
checks the antecedents and the consequents of two questions to determine if they are equivalent.

`Create_Question_Table`
creates a empty question table.

`Save_Question_Table`
saves a question table in files.

`Open_Table`
retrieves a question table and prepares it for reading.

`Delete_Table`
releases the memory used for a question table.

`Store_Question`
stores a question in a question table.

`More_Entries`
determines if there are more questions to be returned using next_entry.

`Next_Entry`
retrieves the next question in a question table.

4.13. Response Schemas

Package Name

Response_Schemas

Description

This package defines the abstract data type Response_Schema. A Response_Schema stores the possible responses for a particular question and the facts to be asserted for each response. Each question has a component of type Response_Schema which defines the response structure of that particular question. This structure must be provided when the question is created. A Response_Schema is made up of structures of type Response_Type, another exported private type. Each Response_Type structure embodies a possible user answer and the facts to be asserted for that answer. There is also is an internal unique identifier for each response which is generated when the response is created. Routines are provided to create and add responses to response schemas, to convert a response schema to a list of answers (to examine them more easily), to retrieve the fact list or Response_Type associated with a given answer, to check an answer for validity with respect to a response schema, to extract the id, answer, and facts from a response type structure, and to test a response schema for emptiness. Persistence routines are also included that are used by higher-level packages to save files of Response_Types.

Objects

```
Answer_Length         : constant Integer := 50;

type Answer_Type is new String (1 .. Answer_Length);
      --This is the type of a single answer.

type Response_Type is private;
      --This type is used to package up information about a single
      --possible response to a question.  It contains the answer itself
      --and the list of facts that are to be asserted for that answer.

type Response_Schema is private;
      --This is the type of the table, stored with each question,
      --which dictates what responses are valid and what facts will
      --be asserted for each response.

type Response_table is private;
      --Response_types are saved to files in the form of response tables

Null_Answer           : constant Answer_Type := Answer_Type'(1 .. 3 => '?',
                                                    others => ' ');
      --value provided for undefined answer

Null_Response         : constant Response_Type;

Null_Response_Schema  : constant Response_Schema;
```

Operations

`Create`
creates an empty response schema.

`Copy`
creates an identical, but separate, copy of its input parameter.

`Add_Response (2)`
this name is overloaded with two operations; one adds a response specified by a fact list and answer value to the response schema table; the other adds an already defined response object to the response schema table.

`Null_and_Free`
replaces a schema with a null schema, freeing any storage occupied by schema contents.

`Null_and_Free_Response`
replaces a response with a null response, freeing any storage occupied by response contents.

`Dump_Schema`
converts a response schema to a list of answers in order to more easily examine the schema.

`Empty`
checks a response schema for emptiness.

`Valid`
checks an answer for validity with respect to a response schema.

`Get_Facts`
retrieve the fact list associated with a given answer.

`Get_Response`
retrieve the Response_Type associated a given answer.

`Get_Id`
extract the identifier from a Response_Type.

`Get_Answer`
extract the answer from a Response_Type.

`Get_Facts`
extract the fact list from a Response_Type.

`Response_Equiv`
returns true if all the components of each of the response parameters are equal, without respect to order.

`Equivalent`
checks all the answers and fact lists, without respect to order, of two response schemas to determine if they are equivalent.

`Create_Response_Table`
creates an empty response table.

`Save_Response_Table`
saves a response table in files.

`Open_Table`
retrieves a response table and prepares it for reading.

`Delete_Table`
releases the memory used for a response table.

`Store_Response`
stores a response in a response table.

`Retrieve_By_Id`
returns a response from a response table with a given id.

`More_Entries`
determines if there are more responses to be returned using next_entry.

`Next_Entry`
retrieves the next response in a response table.

4.14. Rule Bases

Package Name

Rule_Bases

Description

This package defines the abstract data type Rule_Base. A Rule_Base can simply be thought of as a collection of rules. Rule bases can be used to collect rules into logically related groups. This package is generic, and thus it can be used to create any type of rule base. The package also depends on instantiations of the generic package Tau_Lists, which is instantiated with the rule type that is input to this package. A rule base can be created using the routine Create_Rule_Base. It can then be filled with repeated calls to Install. Rules can be deleted from a rule base with Delete. To examine the rules in a rule base, the base can be converted to a rule list (an instantiation of Tau_Lists), using Dump_Base, and then examined using routines provided in the Tau_Lists package. Routines are also provided to determine whether or not a Rule_Base is empty, to change a label, to search for a particular rule in a rule base, to search for a rule given the rule's id, and to extract a rule base's label and id. The set operations union, intersection, difference, and subset are also provided.

Objects

```
type Rule_Base is private;
      --A Rule_Base can be thought of as a collection of rules.

package Rule_Lists is new Tau_Lists (Rule_Type, Rule_Equiv);

subtype Rule_List is Rule_Lists.Tau_List;
      --A list form of a rule base.

Null_Rule_Base : constant Rule_Base;
```

Operations

`Create_Rule_Base`
creates an empty rule base.

`Install`
adds a rule to a rule base.

`Copy`
creates an identical, but separate, copy of its input parameter.

`Null_And_Free_Rule_Base`
nullifies a rule_base releasing any dynamic storage occupied by the base.

`Union`
creates the union of two rule bases.

`Intersection`
creates the intersection of two rule bases.

`Difference`
creates the difference of two rule bases.

`Delete_Rule`
deletes a rule from the rule base.

`Change_Label`
change the label of a rule base.

`Dump_Base`
converts a rule base to a rule list for easier examination of the rules.

`Get_Label`
returns the label of a given rule base.

`Get_Base_Id`
returns the rule base's unique identifier.

`Search_Rule`
searches for a given rule in a rule base.

`Search_By_Id`
searches for a rule with a given id.

`Subset`
determines if one given rule base is a subset of another.

`Empty`
checks a fact base for emptiness.

`Equivalent`
compares all the rules in two rule bases, without respect to order, to see if they are equivalent.

`Save_Labels`
saves the label and identifier of a rule base in a file.

`Initialize_Base`
initializes an empty rule base using the label and identifier retrieved from a file.

4.15. Agendas

Package Name

Agendas

Description

The ADTs defined in this package are information lists, agenda records, and Agendas. These abstract data types comprise the implementation of homogeneous agendas (meaning that all items on the agenda are of the same type). This package is generic, so it can be used to implement several kinds of agendas. Agendas are used to maintain collections of "items", each of which has a weight of some type. Items are placed on the agenda, along with an assigned weight, and then items can be taken off of the agenda in order of their weights. The same item can be put on the agenda more than once. In this case, the weight associated with the item is the sum of the weights assigned each time the item was put on the agenda. The type of the item itself must be input as a generic parameter, as well as a structure, defined by the user, which contains information associated with the act of placing the item on the agenda. Lists of these information structures are maintained along with each of the items on the agenda. For example, an item in a question agenda could be the question itself (or a reference to it), with the information list consisting of the Qrules (or references to Qrules) whose firing led to the posting of the question to the agenda. Another generic parameter is Weight_Type, which is the type of the weights associated with each item. Comparison and addition operators must also be provided for this type. If no operations for Weight_Type are specified, then they default to standard "greater than" and addition for whatever type Weight_Type is. For each exported type, routines are provided to make identical but separate copies of objects, and to compare two objects for equivalence. These routines are meant to provide copy semantics. In order to implement these correctly, routines are needed for making copies of, and testing for equivalence of, objects of the private types that are passed in as generic parameters. These routines must also be provided as generic parameters.

Objects

```
type Info_List is private;
      --An info_list can be thought of as an ordered collection of
      --information structures of type information.

type Agenda_Rec is private;
      --An agenda_rec has three components: the item that the
      --agenda keeps track of, the composite weight of this item,
      --and a list of the information that the user wants to keep
      --track of.

type Agenda is private;
      --An agenda is simply a collection of agenda records.

Null_Info_List   : constant Info_List;

Null_Agenda_Rec  : constant Agenda_Rec;

Null_Agenda      : constant Agenda;
```

Operations

`Create_Agenda`
creates an empty agenda.

`Add_Item`
places an item on the agenda.

`Copy` (3)
this name is overloaded with three operations; one copies an info_list, one copies an agenda_rec and one copies an entire agenda; each creates an identical, but separate, copy of its input parameter.

`Null_And_Free`
replaces an agenda with a null agenda, freeing any dynamic storage occupied by agenda contents.

`Delete_Weightiest_Record`
removes the weightiest agenda record (including an agenda item) from an agenda and returns the record.

`Delete_One_Record`
deletes a given agenda record from an agenda.

`Dump_Agenda`
converts an agenda to a list of agenda items in order to more easily examine the agenda.

`Reset_Info`
prepares an information list for iterating.

`Next_Info`
returns the information structure following the structure returned by the last call to next_info in an information list.

`More_Info`
returns true if there are more information structures that have not been returned by calls to next_info.

`Search_Info`
searches for a given information structure in an information list.

`Get_Item`
extracts the agenda item from an agenda record.

`Get_Weight`
extracts the weight from an agenda record.

`Get_Info`
extracts the information list from an agenda record.

`Search_Record`
returns the agenda record associated with a given agenda item.

`Empty` (2)
this name is overloaded with two operations; one checks whether an info_list is empty and the other checks whether an agenda is empty.

`Equivalent_Rec`
returns true if all the components of the two agenda record parameters are equivalent (using Item_Equiv).

`Equivalent` (2)
this name is overloaded with two operations; separate operations are provided to check equivalence for information lists and agendas.

4.16. Basic AdaTAU Configuration

Package Name

Basic_Configuration

Description

This package brings together in one place all of the data types and operations needed by an inferencer. All of the underlying abstract data types and their associated operations are accessible from this package. Where necessary, generic packages are instantiated to provide the needed data structures. This package also defines procedures to be used by an inferencer for firing rules and for asking questions from an agenda. The firing routines can be used to fire both Irules and Qrules. For each type of rule, routines are provided to determine if the antecedent of the rule is satisfied (or if the rule is "primed"), to fire a rule, to perform both these operations together, and to perform these two operations on each rule in a rule base. The routines associated with asking agenda questions include routines for posing a question, recording an answer, and processing that answer by asserting the facts associated with the answer(s) received.

Objects

```
subtype Tau_Fact_Base is Fact_Bases.Fact_Base;
       -- make basic fact bases type visible to user of this package

package Irule_Bases is new Rule_Bases
       (Irules.Irule,
        Irules.Null_Irule,
        Unique_Identifiers.Uid,
        Irules.Get_Id,
        Unique_Identifiers.Equal);

subtype Irule_Base is Irule_Bases.Rule_Base;
       -- the basic irule_base type via rule_bases generic

package Qrule_Bases is new Rule_Bases
       (Qrules.Qrule,
        Qrules.Null_Qrule,
        Unique_Identifiers.Uid,
        Qrules.Get_Id,
        Unique_Identifiers.Equal);

subtype Qrule_Base is Qrule_Bases.Rule_Base;
       -- the basic qrule_base type via rule_bases generic

package Question_Bases is new Rule_Bases
        (Questions.Question,
         Questions.Null_Question,
         Unique_Identifiers.Uid,
         Questions.Get_Id,
         Unique_Identifiers.Equal);

subtype Question_Base is Question_Bases.Rule_Base;
       -- the basic question_base type via rule_bases generic
```

```
package Q_Agendas is new Agendas (Unique_Identifiers.Uid,
                                  Qrules.Qrule,
                                  Integer, Copy_Uid,
                                  Unique_Identifiers.Equal,
                                  Qrules.Copy,
                                  Qrules.Equivalent);

subtype Question_Agenda is Q_Agendas.Agenda;
      -- the basic question_agenda type via the agendas generic

package User_Responses is new Tau_Lists (Response_Schemas.Response_Type);

subtype Tau_Response is Fact_Lists.Fact_List;
      -- make a repsonse type visible to the user of this package
```

Operations

`Primed` (2)
: this name is overloaded with two operations; one operation determines if an irule is ready to fire, the other determines if a qrule is ready to fire.

`Fire` (2)
: this name is overloaded with two operations; one fires an irule by asserting its consequent facts, and the other fires a qrule by placing a question on the agenda.

`Prime_And_Fire` (2)
: this name is overloaded with two operations; each operation checks that a rule (irule or qrule respectively) is ready to fire, and then, if it is, fires it.

`Fire_Base` (2)
: this name is overloaded with two operations; for each rule in the rule base (irule_base or qrule_base respectively), it is first determined if the rule is ready to fire and then, if it is, the rule is fired.

`Pose_Question`
: poses a question to the user.

`Record_Response`
: takes an answer or answers from the user.

`Process_Response`
: asserts the facts associated with each answer.

4.17. Generic Advanced AdaTAU Configuration

Package Name

Advanced_Configuration

Description

This generic package brings together in one place all of the data types and operations, beyond those provided in the Basic_Configuration, needed to support a distributed TAU-style inference system. All of the new underlying abstract data types and their associated operations are accessible from this package. Where necessary, generic packages are instantiated to provide the needed data structures. This package also defines procedures to be used by an inferencer for firing rules and for dealing with the focus agenda. In particular, this package defines the abstract data type inference context. An inference context collects all of the individual rule bases, as well as a local agenda maintaining the state of interaction with the user, so that a Think - Ask - Update style inference scheme can be applied to multiple distributed rule bases. The basic rule base identities established for a centralized version of AdaTAU are imported from Basic_Configuration. This package defines an frule base used to provide rule base context switching, as well as the inference context definition itself. There is also is an internal unique identifier for each inference context which is generated when the inference context is created. Routines are provided to create an inference context, to extract the various components from an inference context, and to change the label of an inference context We define an operation to fire FRules. As in the case of IRules and QRules, routines are provided to determine if the antecedent of the rule is satisfied (or if the rule is "primed"), to fire a rule, to perform both these operations together, and to perform these two operations on each rule in a rule base. The routines associated with the focus agenda provide for evaluating the agenda to see if a pending focus switch should be processed, packaging up the data structures providing the state of the local inference process, and decomposing an inference state description as extracted from the global agenda.

Objects

```
package Frules_Inst is new Frules (Context_Reference_Type,
                                   Null_Context_Reference, Copy_Reference,
                                   Equal_Reference);


subtype Frule is Frules_Inst.Frule;

package Frule_Bases is new Rule_Bases (Frule, Frules_Inst.Null_Frule,
                                       Unique_Identifiers.Uid,
                                       Frules_Inst.Get_Id,
                                       Unique_Identifiers.Equal,
                                       Frules_Inst.Equivalent);

subtype Frule_Base is Frule_Bases.Rule_Base;

type Inference_Context is private;
   --A context includes the associated rule bases as well as question
   --base necessary to provide for an inference process to take place
```

```
package F_Agendas is new Agendas
       (Context_Reference_Type, Frules_Inst.Frule_Switch_Context, Integer,
        Copy_Reference, Equal_Reference, Frules_Inst.Copy_Frule_Switch_Context,
        Frules_Inst.Equal_Frule_Switch_Context);

subtype Focus_Agenda is F_Agendas.Agenda;

package F_Suggestions is new Tau_Lists (Frules_Inst.Frule_Switch_Context);

subtype Focus_Suggestions is F_Suggestions.Tau_List;

Null_Context          : constant Inference_Context;

Null_Focus_Suggestion : constant Focus_Suggestions :=
      F_Suggestions.Null_List;
```

Operations

`Create`
creates a new inference context with the given components.

`Copy`
returns a inference context which is identical to, but separate from, context.

`Put_Context_Reference_Id`
inserts a context reference id into an inference context.

`Put_Imports`
inserts an import fact parameter list into an inference context.

`Put_Exports`
inserts an export fact parameter list into an inference context.

`Put_Irule_Base`
inserts an irule base into an inference context.

`Put_Qrule_Base`
inserts a qrule base into an inference context.

`Put_Frule_Base`
inserts an frule base into an inference context.

`Put_Question_Base`
inserts a question base into an inference context.

`Put_Fact_Base`
inserts a fact base into an inference context.

`Put_Schema`
inserts a fact base schema into an inference context.

`Put_Local_Agenda`
inserts a question agenda into an inference context.

`Update_Agenda`
update the agenda for the context with a new agenda.

`Get_Context_Reference_Id`
returns the context's application id.

`Get_Imports`
returns the context's import fact parameter list.

`Get_Exports`
returns the context's export fact parameter list.

`Get_Table_Id`
returns the context's table identifier.

`Get_Irule_Base`
returns context's irule_base.

`Get_Qrule_Base`
returns context's qrule_base.

`Get_Fact_Base`
returns context's local fact base.

`Get_Question_Base`
returns context's question_base.

`Get_Frule_Base`
returns context's frule_base.

`Get_Schema`
returns context's fact base schema.

`Get_Local_Agenda`
returns context's local question agenda.

`Equivalent`
returns true if all of the components of the two context parameters are equivalent.

`Primed`
attempts to satisfy the antecedent of rule. It searches fbase for each of the facts in rule's antecedent. If it successfully finds them all, then it returns true.

`Fire`
fires rule by adding the associated context switch suggestion to context_switch_suggestions.

`Prime_And_Fire`
first checks rule to see if it is primed and then, if it is, fires it.

`Fire_Base`
for each rule in rbase, the antecedent of the rule is first checked, using fbase, and then, if the antecedent can be satisfied, the rule is fired.

4.18. RLF Instance of Advanced AdaTAU Configuration

Package Name

Librarian_Configuration

Description

This package declares an application-specific instance of the advanced_configuration package. In particular, a specific type: a context identity is passed as a generic parameter whose values serve to identify particular inference contexts within a larger application specific data structure. This particular example is designed to provide an interface between the AdaKNET subsystem and AdaTAU where inference contexts are associated with generic concepts within an AdaKNET network. The Adanet_Object_Name_Type from the Adanet_Name_Types package serves as the context identity for the Librarian_Configuration.

Objects

```
package Lib_Hybrid is new Advanced_Configuration
       (Adanet_Name_Types.Adanet_Object_Name_Type,
        Adanet_Name_Types.Null_Adanet_Object_Name,
        Adanet_Name_Types.Copy_Adanet_Object_Name_Type,
        Adanet_Name_Types.Equal_Adanet_Object_Name_Type);

subtype Lib_Frule is Lib_Hybrid.Frule;

subtype Lib_Frule_Base is Lib_Hybrid.Frule_Base;

subtype Lib_Inference_Base is Lib_Hybrid.Inference_Context;

subtype Lib_Focus_Agenda is Lib_Hybrid.Focus_Agenda;

Null_Lib_Inference_Base : constant Lib_Inference_Base :=
      Lib_Hybrid.Null_Context;

subtype Lib_Inference_Base_Id is Adanet_Name_Types.Adanet_Object_Name_Type;

subtype Lib_Fact_Parameter_List is Fact_Parameter_Lists.Fact_Parameter_List;
```

4.19. Component Persistance Management

Package Name

```
Persistence
```

Description

This package contains routines to store and retrieve AdaTAU data objects to and from files. Routines are provided to save and restore fact bases, fact base schemas, Irule bases, Fact Parameter Lists Qrule bases, Frule bases, question bases, and agendas, as well as to save an entire AdaTAU session. This package also renames and re-exports various Free operations to manage memory.

Objects

```
subtype Inferencer_Name_Type is String (1..80);
        -- Provides name type to associate with named files
```

Operations

`Is_Saved`
returns true if inferencer_name corresponds to an inferencer that has been previously saved.

`Save_Labels (2)`
saves labels and unique identifiers corresponding to fact base schemas and fact bases respectively to a file.

`Save_Fact_Base`
saves a fact base and its associated fact base schema.

`Save_Fact_Base_Schema`
saves a fact base schema.

`Save_Irule_Base`
saves an irule base.

`Save_Qrule_Base`
saves a qrule base and its associated question base.

`Save_Frule_Base`
saves an frule base.

`Save_Question_Base`
saves a question base.

`Save_Agenda`
saves a local question agenda.

`Save_Fact_Parameter_Lists`
saves the import and export fact parameter lists.

`Save_Inferencer`
saves an irule base, a qrule base, a question base, and an agenda.

`Initialize_Base`
initializes an empty fact base using a label and identifier retrieved from a file.

`Initialize_Schema`
initializes an empty fact base schema using a label and identifier retrieved from a file.

`Load_Fact_Base`
loads a fact base and its associated fact base schema.

`Load_FBase_Schema`
loads a fact base schema.

`Load_Irule_Base`
loads an irule base.

`Load_Qrule_Base`
loads a qrule base and its associated question base.

`Load_Question_Base`
loads a question base.

`Load_Agenda`
loads a local question agenda.

`Load_Frule_Base`
loads an frule base.

`Load_Fact_Parameter_Lists`
loads the import and export fact parameter lists.

`Load_Inferencer`
loads an irule base, a qrule base, a question base, and an agenda.

`Restart_Inferencer`
loads the irule base, qrule base, question base, and agenda from files.

`Free_Fact_Base`
makes fact base a null fact base and releases any storage occupied by the fact base.

`Free_Fbase_Schema`
makes fact base schema a null fact base schema and releases any storage occupied by the fact base schema.

`Free_Irule_Base`
makes irule base a null irule base and releases any storage occupied by the irule base.

`Free_Qrule_Base`
makes qrule base a null qrule base and releases any storage occupied by the qrule base.

`Free_Question_Base`
makes question base a null question base and releases any storage occupied by the question base.

`Free_Agenda`
makes agenda a null question agenda and releases any storage occupied by the question agenda.

`Free_Inferencer`
releases all storage occupied by the irule base, qrule base, question base, fact base, and question agenda.

`Free_Frule_Base`
makes frule base a null frule base and releases any storage occupied by the frule base.

`Free_Fact_Parameter_Lists`
makes Imports and Exports Null fact parameter lists.

4.20. Basic AdaTAU Persistance Management

Package Name

`Static_Persistence`

Description

This package provides high-level load/save operations for AdaTAU inferencers.

Objects

```
subtype Inferencer_Name_Type is Persistence.Inferencer_Name_Type;
```

Operations

`Save_Inferencer`
saves all the input objects in files.

`Load_Inferencer`
loads the irule base, qrule base, question base, and fact base schema from files.

`Free_Inferencer`
releases the memory occupied by the irule base, qrule base, question base, and fact base schema.

4.21. Distributed AdaTAU Persistance Management

Package Name

```
Lib_Static_Persistence
```

Description

This package provides high-level load/save operations for Librarian/distributed AdaTAU inference bases.

Objects

```
subtype Inference_Base_Name_Type is
      Adanet_Name_Types.Adanet_Object_Name_Type;
```

Operations

`Save_Inference_Base`
saves an inference base in files.

`Load_Inference_Base`
loads an inference base from files.

`Free_Inference_Base`
releases the memory occupied by an inference base.

4.22. Reverse RBDL Translator

Package Name

```
Dump_Tau_Components
```

Description

This package provides operations to generate a listing of an AdaTAU knowledge base consisting of rule bases, question base, fact base schema and fact base to the currently assigned output file (the operator's console by default). This listing is produced in RBDL format, suitable for input to the RBDL processor for initializing the internal representation required to execute AdaTAU.

Objects

```
-- none exported by this package
```

Operations

`Dump_Fschema`
list a single fact schema (normally as part of a fact base schema).

`Dump_Fb_Schema`
list an entire fact base schema.

`Dump_Fact`
list a fact (normally as part of a fact list included in a rule description).

`Dump_Flist`
list an entire list of facts (normally as part of a rule description).

`Dump_Fbase`
list an entire fact base.

`Dump_Irule`
list an IRule description (normally as part of a rule base description).

`Dump_Irbase`
list an entire IRule base.

`Dump_Rschema`
list a response schema (normally as part of a question description)

`Dump_Question`
list a question (normally as part of a question base, or an individual QRule).

`Dump_Qrule`
list a QRule description (normally as part of a rule base description).

`Dump_Qrbase`
list an entire QRule base.

`Dump_Qbase`
list an entire question base.

`Dump_All`
provide a complete RBDL description of an AdaTAU knowledge base including IRule base, QRule base, question base, fact base schema, and fact base.

4.23. AdaTAU Inference Cycle Components

Package Name

```
Tau_Cycle_Components
```

Description

This package provides the three subprograms that implement the Think - Ask - Update operations. These operations are basic to the rule-based inference approach planned for the Reusability Library Framework. Rules must be provided as distinct IRule and QRule bases. All of the operations exported here are composite operations built from primitives supplied in the basic_configuration. The structure of these composites is defined by the inference mechanism summarized by Think - Ask - Update.

Objects

```
-- no new objects are exported
```

Operations

`Think`

provides for a forward-chained generation of facts based on a current fact base, and supplied rule bases. When Think returns control to its caller, no firable rules exist in either rule base, and no further modifications to the fact base are possible pending processing of the agenda.

`Ask`

handles interaction with a user, based on an agenda of user queries which are scheduled as a result of rules fired during the Think phase. Ask delivers a response object that will be processed by Update.

`Update`

modifies the fact base based on a user's response recorded during Ask. Update also provides truth maintenance to keep the fact base consistent after user derived modifications have been processsed.

4.24. Distributed AdaTAU Inference Cycle Components

Package Name

```
Dtau_Cycle_Components
```

Description

This package provides several additional subprograms that along with the Think - Ask - Update operations, provide the basic services of the distributed, rule-based inference approach used by the Reusability Library Framework. In particular, these operations provide for the processing of focus rules, and handling of focus switches to other rule bases.

Objects

```
-- no new objects are exported
```

Operations

`Think_After`
provides for a forward-chained generation of facts based on a current fact base, and supplied rule bases. When Think_After returns control to its caller, no firable rules exist in either irule or frule bases, and no further immediate modifications to the fact base are possible before either processing the local agenda or performing a context switch.

`Acquire_Tau_Context`
The inference base parameter indirectly provides a handle to the individual TAU components that enable a local investigation to be begun or continued. Acquire_TAU_Context translates this handle to the actual object instances required.

`Save_Tau_Context`
provides a means of depositing the local state information about a local investigation just before a context switch is about to be made. In particular, the local agenda is likely to have changed since the current investigation was begun or resumed. Save_TAU_Context uses the inference base handle to save information that can be recalled later.

`Eval_Focus`
From the focus agenda, Eval_Focus will produce the current top two choices for further local investigation.

4.25. AdaTAU Basic Inference

Subprogram Name

Tau

Description

The Tau subprogram provides a basic entry point to the rule base services provided by AdaTAU. The main modules employed by TAU (Think, Ask, Update) are themselves encapsulated in a separate package (Tau_Cycle_Components). The rule base facilities that are defined by AdaTAU are made available through the basic_configuration package.

Tau

is organised according to the Think - Ask - Update paradigm. This approach delays user interaction as long as forward-chained progress occurs using the available rules (Think). User interaction is managed through the use of an agenda structure. User responses are processed by one module (Ask) while consequent changes to the fact base are handled by another (Update). The basic loop continues until the agenda is exhausted, and no further fact derivations are possible.

4.26. Distributed AdaTAU Inference

Subprogram Name

`Dtau`

Description

The DTAU subprogram provides a basic entry point to the distributed rule base services provided by AdaTAU. The main modules employed by DTAU are themselves encapsulated in a separate package (DTAU_cycle_components). The rule base facilities that are defined by AdaTAU are made available through the basic_configuration package as well as an additional package of basic services for use in a distributed setting: an instantiation of advanced_configuration.

`Dtau`

provides a skeletal distributed rule base inference plan that expands upon that provided in standard TAU. From the current global agenda, DTAU determines an initial TAU context in which to begin. A basic TAU sequence is invoked, refocus suggestions produced after any primed FRules are fired are merged with the global agenda, and an inference base at which to continue inferencing is determined. Focus switches take place when a global agenda item is at the head of the global agenda and there is no data that suggests this pending context switch should be ignored. After each such context switch, the basic TAU sequence is executed at least once. The current implementation supports an eager surrender strategy where any pending context switch is executed regardless of the local agenda. The basic loop continues until both the current local agenda as well as the focus agenda are exhausted.

5. AdaTAU Specification Language — RBDL

The AdaTAU inferencer manipulates many different objects of different types. The different types of objects are described in the Abstract Data Types description in this document and by the Ada package specifications themselves. Before the inferencer can begin the Think-Ask-Update process, instances of these objects must be created, tailored, and initialized. Operations are provided to do this, but the knowledge engineer must guide this process by specifying the numbers and types of objects to be created, specifying various properties which tailor the objects to a specific application, and supplying the values which are to be used to initialize the objects. The RBDL (Rule Base Definition Language) provides a simple vehicle for the knowledge engineer to use to accomplish this task. The specification language is used to specify exactly what objects are to be created, to manipulate various properties of these objects, and to give them initial values, where appropriate. The RBDL processor translates the RBDL specification provided by the knowledge engineer into calls to routines in the ADT package specifications which in turn create and manipulate the objects.

RBDL specifications are translated into an executable Ada procedure by the RBDL processor program. This program is itself written in Ada and when compiled provides the RBDL translation capability. The RBDL processor program is included with this version of AdaTAU.

RBDL provides the means to specify the content of inference bases that are used to conduct a TAU-style inference directed toward manipulating and updating a global fact base. RBDL also supports the use of multiple inference bases that communicate with each other through the use of fact parameters and the firing of focus rules. RBDL specifications of the state of an AdaTAU session can also be generated by AdaTAU itself to stand as a basis for comparison between a start-up state for an AdaTAU session, and the final state after no further inference progress is possible. Note that the current version of distributed AdaTAU does not support the generation of RBDL corresponding to the state of a distributed inference session. Using an editable inference base description, it will be possible to run AdaTAU from the point it left off, after appropriate new rules or facts are entered directly into the saved RBDL specification.

The Backus-Naur Form of the RBDL syntax is provided in Appendix A. The syntax was designed to resemble Ada code as much as possible. RBDL provides a declarative syntax for the specification of rule bases and fact bases. RBDL specifications are themselves translated to calls on specific routines that are included in the various AdaTAU ADT packages which are described in the implementation level section. An AdaTAU RBDL specification consists of a set of definitions that are of several basic types: initial fact base definitions, fact base schema definitions, question base definitions, rule base definitions, inferencer definitions and fact parameter definitions. A single specification can contain any number of question base or rule base definitions. However, an RBDL specification file may contain only one instance of a fact base schema definition or an initial fact base definition.

Fact Base Definitions

A fact base can be defined by providing an identifier for it, an identifier specifying a fact base schema that will define the structure of the fact base, and the facts to be used to

initialize the fact base. The fact base schema identifier must correspond to a previously defined fact base schema and the facts must be consistent with this schema. The RBDL processor will use the "Create" function provided in the fact bases package to create the new fact base and to associate the specified fact base schema with it. Then the "Install" routine will be used to put the specified facts into the fact base, one at a time. If the knowledge engineer wishes to define an empty fact base, the keyword "null" can be used in place of the list of initial facts.

Fact Base Schema Definitions

A fact base schema definition basically consists of a set of attribute definitions. For each attribute, the knowledge engineer must specify a name, a type, and a list of values that can be associated with that attribute. All the attribute names must be unique within the fact base schema. The type of an attribute indicates whether the attribute can take on just one of the listed values, any number of the listed values, or any value at all (in the last case, no value list should be provided). The list of values is simply a list of text strings. If several different attributes have the same type and the same value list, then the names of these attributes can be grouped together and the type and value list specified just once. Of course, an identifier must be provided for the fact base schema as well. To create a fact base schema, the RBDL processor will create a structure of the type fact_base_schema, provided in the fact_base_schemas package, and initialize it with other operations from this package.

Question Base Definitions

To define a question base, the knowledge engineer need only provide a list of question definitions. For each question, an identifier, the text of the question, its type, and the list of possible responses must be provided. The type of a question is similar to the type of an attribute. It indicates if the question can be answered with exactly one of the listed responses, any number of the listed responses, or with any response at all (in the last case, no response list should be provided). For each response in the response list, the text of the response must be provided, as well as a list of facts that will be asserted when this response is received.

The RBDL processor will first use the "Create" routine in the Question_Bases package (an instantiation of Rule_Bases) to create an empty question base. Then, for each question specified, a variable of type response_schema, provided in the response_schemas package, will be created and initialized. The fact lists that are part of the response schema will be created using "Create" and "Build_List" from the Fact_Lists package (which is an instantiation of the Tau_Lists generic package). This structure will then be used by another "Create" routine in the questions package to create a question. Finally, the question will be put in the question base using "Install" from the Question_Bases instantiation of Rule_Bases.

Rule Base Definitions

Rule bases, and thus rule base definitions, come in three types: IRule (inference rule) bases, QRule (question-asking rule) bases, and FRule (focus rule) bases. Either type of definition must provide an identifier that names the inference base. An IRule base definition must also provide a list of IRule definitions, each of which consists of two

lists of facts (representing the antecedents and consequents) and a textual justification for the rule. Similarly, a QRule base definition consists of a list of QRule definitions. Each QRule definition must include a list of facts representing the antecedents of the rule, an identifier referring to a question (which must be previously declared as part of a question base definition), a numerical weight which will be associated with the question when it is put on an agenda, and a textual justification for the rule. Frules are structured like QRules except that instead of a question identifier, an FRule declares a focus identifier that names another inference context. An FRule may optionally name an export fact list whose contents are added to a fact list which is used in conjunction with fact parameters to control how information is passed between cooperating inference contexts.

To create an empty rule base, the "Create" routine from the appropriate instantiation of the Rule_Bases package will be used. Then each rule will be created using another "Create" routine in either the IRules, QRules or FRules package. Again, the fact lists that are part of the rules will be created using "Create" and "Build_List" from the fact_lists package. Finally, each rule will be will be put in the rule base using "Install".

Fact Parameter Definitions

Within an inferencer definition, fact parameters are identified by name as well as parameter class within distinct inference contexts. The collection of local contexts defines a global inference environment and fact parameters provide a method by which information (facts) are exchanged between individual local inference bases. Within an inference context description, a complete list of fact parameters for the local context is declared, where the list is organized according to whether the parameter is imported or exported.

Using RBDL

From a RBDL specification of a local inference context, the RBDL processor produces an Ada procedure called Initialize_TAU_Components which includes the necessary calls on the operations provided within AdaTAU to build the required rules bases, question base, fact base schema and initial fact base, and make these persistent for use by an application needing rule base services. The body of the procedure makes all the necessary translations of the parameters to underlying data structures and makes any necessary initializations. This procedure is embedded in an Ada library unit (main program) which prepares the resulting fact base for use by the application. More information on actually using RBDL to create an application-specific rule base description and integrating it with an application that uses AdaTAU's programmatic interface is given in the next section.

The use of RBDL to support distributed inference base description and processing is also possible. In particular, the RBDL processor can be used to process multiple inference bases individually, and a client application can use the DTAU procedure to initiate inferencing at one of them.

6. Using AdaTAU

In order to use AdaTAU effectively, it is advisable to prepare RBDL specifications for the fact and rule base components that AdaTAU is equipped to process. One can directly make use of primitive AdaTAU operations to define the necessary knowledge base components, but to do so requires handcoding a large number of Ada procedure calls. RBDL is provided so that a rule-based application can be constructed declaratively, using a template main program that requires few, if any, modifications. This section describes how one can start with a template main program and a RBDL specification file, and produce a working Ada program that is able to interact with the user and make corresponding inferences based on the available facts and rules.

If the AdaTAU subsystem is to be embedded in a larger Ada application, then direct calls to the relevant AdaTAU operations must be provided by the application designer. Programmers wishing to use AdaTAU in a larger application should consult the ADT descriptions given in section 4 of this manual. AdaTAU is targeted for these sorts of embedded applications and the stand-alone example given in this section is meant to

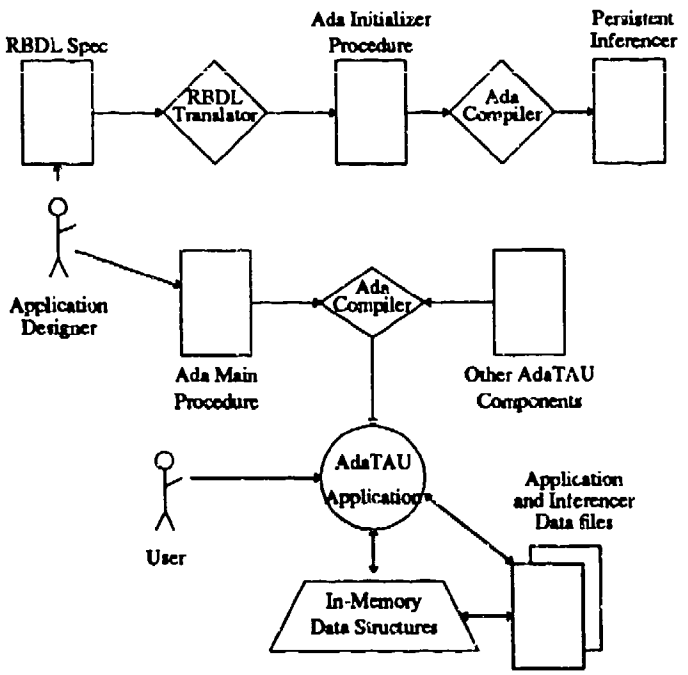


Figure 5. Creating an AdaTAU Application

show the integration methodology to combine statically declared rule base components with Ada code that processes internal forms of derived fact and rule bases. To show that this internal processing is performed correctly, the example invokes a handwritten reverse RBDL translator that writes out RBDL representations of the final forms of the derived fact and rule bases. A complete transcript of an interactive session to produce and run an AdaTAU application is presented at the end of this section.

The necessary steps to build a working AdaTAU application are elaborated in the next three subsections. These steps can be summarized as follows (see figure 5).

1) Use an ordinary text editor to prepare a RBDL specification file. This file contains a description of rule bases, question bases, a fact base schema and an initial fact base (if any) that support the domain.
2) Execute the RBDL processor on the specification file to yield an Ada procedure capable of initializing all necessary components required before AdaTAU operations can be applied to the domain captured in the RBDL specification.
3) Compile and execute the RBDL output to produce a persistent form of the RBDL inferencer for use by the application.
4) Using Ada `with` clauses as appropriate, create an Ada main program which contains a call to the AdaTAU persistence routine that brings an inferencer into memory for processing by AdaTAU. Include references to any other AdaTAU operations that are required by the application. Typically, one of these operations is the TAU procedure itself which provides the *Think — Ask — Update* inference mechanism (or DTAU if the application makes use of multiple, distributed inferencers). The Dump_RBDL operation is useful to check that all of the expected rule bases, as well as a final fact base, have been installed and processed correctly.
5) Compile and execute the desired application program. The user interacts with the program, and depending on the AdaTAU components chosen, persistent versions of the in-memory data structures can be saved and restored from disk files.

Using the facilities of distributed AdaTAU requires that steps 1, 2 and 3 be repeated for each of the localized inference bases anticipated by the application designer. In addition, the application designer must provide an application-specific scheme for locating the various inference bases within the applications own data structures. The interested reader is advised to consult the RLF Librarian user manual for a discussion of one method of distributing and coordinating the use of multiple inference bases. The rest of this section assumes that only a single inference base is under consideration, so that none of the facilities of RBDL that support distribution of, and communication between, inference bases are relevant.

6.1. Creating AdaTAU Knowledge Bases

Appendix B contains an extended, non-distributed RBDL example showing each of the major RBDL divisions. Whenever AdaTAU is to be applied to a new domain, it is necessary to consider the essential information that is to be tracked and maintained about this domain and the rules by which new information about this domain is to be deduced. The example captures some relevant information concerning the taxonomy of general

Ada components presented by Grady Booch in his recent book [Booch87].

This RBDL knowledge base is prepared by entering it into the system text editor. The preparer of a RBDL specification should be careful to obey the syntactic rules of RBDL so that the RBDL processor can smoothly translate the specification into the necessary Ada calls. However, if a mistake is made in the RBDL specification, an error message is reported on the user's console with a line number indicating the approximate location of the offending statement. The RBDL processor is currently limited in the way such errors are reported and a new user may find that only the first error is reported even when the RBDL source contains several such errors. When the RBDL processor finds no syntax errors, the RBDL processor performs some semantic error checking on the RBDL specification. The current version of the RBDL processor performs minimal semantic error checking. Semantic errors not caught at generation time will be caught at run-time by the appropriate AdaTAU routines. It is expected that later versions of the RBDL processor will provide more robust semantic error checking with a corresponding lessening of reliance on AdaTAU run-time error checking.

RBDL is invoked simply by executing

```
rbdl < -file_name-
```

where -file_name- is the name of file containing the RBDL source. If there are no errors, rbdl produces an Ada source file called rbdlprog.a. This file contains an Ada procedure `Create_Inferencer` which includes a local subprocedure with header:

```
procedure Initialize_TAU_Components
     (FB_Schema      : out Fact_Base_Schemas.Fact_Base_Schema;
      Question_base  : out Basic_Configuration.Question_Base;
      Irule_Base     : out Basic_Configuration.Irule_Base;
      Qrule_Base     : out Basic_Configuration.Qrule_Base;
      Frule_Base     : out Librarian_Configuration.Lib_Frule_Base;
      Initial_Fact_Base : out Fact_Bases.Fact_Base;
      Import_Fact_list : in out Fact_Parameter_Lists.New_Lists.Tau_List;
      Export_Fact_list : in out Fact_Parameter_Lists.New_Lists.Tau_List) is
```

In addition, RBDL generates the necessary Ada `with` clauses to enable this procedure to be compiled in the context of other necessary AdaTAU source files. This file should be compiled with the entire AdaTAU library visible during the compile process. After the user compiles and executes the `Create_Inferencer` main program, a persistent form of the AdaTAU inferencer defined by the RBDL specification is created that is then processed by the application as required. The application program that makes use of the RBDL rule base specification should include a `with` clause referencing the necessary operations to bring the inferencer into main memory and to initialize the necessary run-time data structures.

6.2. Building an AdaTAU Application

Once the RBDL-specified AdaTAU knowledge base has been created by running it through the RBDL processor, the application-specific code must be written and compiled. In its simplest form, an application must contain a call on the routine in the AdaTAU persistence package to bring an inferencer into memory for processing. Alternatively, if an inferencer is being developed to be used in the context of a number of distributed inference bases, the name of the inferencer can be used to locate and retrieve an

inferencer for processing by the DTAU procedure. The following sample main program simply illustrates this process, and includes a minimal amount of extra code.

```
with Basic_Configuration, Fact_Lists,
     Dtau, Dump_Tau_Components,
     Fixed_Strings, Commonio, Lib_Static_Persistence, Librarian_Configuration;

use Basic_Configuration;

procedure Rbdl_DTAU_Test_Driver is

   use Commonio;

   Working_Fact_Base : Tau_Fact_List := Fact_Lists.New_Lists.Create_Tau_List;
   Global_Ag         : Librarian_Configuration.Lib_Focus_Agenda :=
                 Librarian_Configuration.Lib_Hybrid.F_Agendas.Create_Agenda;
   Explain_DTAU      : Librarian_Configuration.Lib_Hybrid.Explain_DTAU_record
                          := (true,true,true,false);
   Context_Name : Lib_Static_Persistence.Inference_Base_Name_Type;
   Len          : Integer;

begin

   Commonio.Put_Message
       ("What context should this test start with?");
   Commonio.Put_Message ("(please use all lower case characters)");
   Commonio.Prompt;
   Context_Name := Fixed_Strings.Pad (Commonio.Get_String, Context_Name'LENGTH);

   DTau (Working_Fact_Base, Global_Ag, Context_Name, Explain_DTAU);

   Dump_Tau_Components.Dump_Flist (Working_Fact_Base);

end Rbdl_DTAU_Test_Driver;
```

Note that in this example, the Working_Fact_Base is initially empty. However, any initial fact base that was saved when the inferencer was created will be restored as DTAU begins execution. The basic DTAU-style inference procedure will be executed using the internal forms of the individual fact, question and rule bases. Finally, a simple dump of the resulting facts will be written by the procedure Dump_Flist found in the included AdaTAU package Dump_Tau_Components.

Neither of the latter routines is required; however, some method must be employed to create internal forms of the required AdaTAU objects. Any of the individual AdaTAU operations can be executed by the application, and the information produced by these operations (typically new facts) can be "harvested" in a manner consistent with the application. The execution of Tau or Dtau can change only the fact base component so that the other AdaTAU components will remain unchanged from their initial state.

6.3. Sample Session

This subsection shows a sample transcript, using the Verdix Ada Development System (VADS) on a Sun workstation, that illustrates the process of creating the beginnings of a distributed AdaTAU-based application. Before executing an RLF tool, the UNIX environment variable RLF_LIBRARIES must be set as appropriate. Consult the Version

Description Document for information about the use of environment variables within the RLF.

Only one inferencer will be created during this script. Other inferencers can be created by following the steps shown in this section. It is assumed that a text editor has been used to prepare a RBDL specification file (appendix B was used in preparing this sample session). The application-specific main program must also have been prepared (the sample from the preceding section was used for this transcript). User entered text follows the system host name prompt (% in this example). Explanatory remarks about the transcript are given in lines begun with the Ada comment delimiter "--".

```
-- Run the RBDL processor on a RBDL source file.

% rbdl < booch_example.rbdl

-- RBDL responds as follows (when there are no errors in RBDL source).

Parsing input.
Parsing completed successfully.
Entering attribute evaluation phase.
Exiting attribute evaluation phase.

-- RBDL generates an Ada source file which is transferred to the directory
-- in which the RBDL application is to be compiled.

-- Compile the generated RBDL source file in a properly initialized
-- Ada library.

% ada -v rbdlprog.a

-- The name of the main subprogram is create_inferencer.  The main
-- Ada library containing the AdaTAU files which the current Ada library
-- refers to must include a reference to the interrupts.o object file
-- included in the RLF distribution.

% a.ld create_inferencer interrupts.o

-- Next, the inferencer is made persistent by executing the object file.

% a.out

-- Using the make facility provided in VADS, build the desired AdaTAU
-- application using the required AdaTAU source files.
-- In this example, a simple test harness is being generated.
-- The top-level procedure is called Rbdl_DTAU_Test_Driver and the
-- resulting object file is called DTAU_test.
-- The full AdaTAU library must be available in this directory in order
-- to build the application.

% a.make -v Rbdl_DTAU_Test_Driver -o DTAU_test -f dist_rbdl_test.a

-- VADS responds as follows, showing successful compilations and final
-- link.

finding dependents of: dist_rbdl_test.a
compiling dist_rbdl_test.a
```

```
      body of rbdl_dtau_test_driver
      spec of rbdl_dtau_test_driver
/ada/vads5.5/bin/a.ld rbdl_dtau_test_driver -o DTAU_test

-- The user now executes the generated program

% DTAU_test

-- Respond to the prompt with the name of the inferencer to begin with.

What context should this test start with?
(please use all lower case characters)
> booch_taxonomy

-- booch_taxonomy is the name of the inferencer as given in the RBDL file.

-- Note that in the display below, the explanations provided by the Test
-- program are not shown to reduce the included output from the test run.

-- The example application walks the user through a portion of Grady
-- Booch's component taxonomy, asking questions as scheduled through
-- the firing of QRules.  The user eventually specifies one of the
-- available Booch components.

What is the component type?


          1.  structure
          2.  tool
          3.  subsystem

Type in the number of your response:
> 2
Is the tool a


          1.  utility
          2.  filter
          3.  pipe
          4.  sorting tool
          5.  searching tool
          6.  pattern matching tool
          7.  other

Type in the number of your response:
> 2
Are the semantics of the component preserved when more than one
thread of control exists?


          1.  yes
          2.  no

Type in the number of your response:
> 1
Does the component guarantee mutual exclusion, or does the user have
to ensure mutual exclusion
```

```
          1.  mutual exclusion guaranteed
          2.  user must provide mutual exclusion

Type in the number of your response:
> 2
Do objects that are part of the component have a size that is static
throughout the object's life, or does the size change dynamically?


          1.  size is static
          2.  size changes dynamically

Type in the number of your response:
> 2
Does the component take care of garbage collection?


          1.  yes
          2.  no

Type in the number of your response:
> 1
Is an iterator provided for this component?


          1.  yes
          2.  no
          3.  not applicable

Type in the number of your response:
> 2

-- After the interaction is finished, the test application dumps out the
-- RBDL specification, as it was originally entered, along with the final
-- fact base that is generated during the interaction.
-- This output is truncated in this transcript after the fact base.

FACT BASE SCHEMA  IS
   component_type: ONE_OF
      ( structure,
      tool,
      subsystem);
   granularity: ONE_OF
      ( monolithic,
      polylithic);
   structure_type: ONE_OF
      ( stack,
      queue,
      string,
      deques,
      rings,
      maps,
      sets,
      bags,
      lists,
```

```
      trees,
      graphs,
      other);
   tool_type: ONE_OF
      ( utility,
      filter,
      pipe,
      sorting,
      searching,
      pattern_matching,
      other);
   multiple_threads: ONE_OF
      ( true,
      false);
   mutex_provided: ONE_OF
      ( true,
      false);
   multiple_readers: ONE_OF
      ( true,
      false);
   static_size: ONE_OF
      ( true,
      false);
   garbage_collection_provided: ONE_OF
      ( true,
      false);
   controlled: ONE_OF
      ( true,
      false);
   iterator_provided: ONE_OF
      ( true,
      false);
   form1: ONE_OF
      ( sequential,
      guarded,
      concurrent,
      multiple);
   form2: ONE_OF
      ( unbounded,
      bounded);
   form3: ONE_OF
      ( unmanaged,
      managed,
      controlled);
   form4: ONE_OF
      ( iterator,
      noniterator);

END ,

-- The following fact base is a result of executing the basic TAU
-- inferencer; the actual initial fact base obtained from the REDL
-- specification was null.

INITIAL FACT BASE init_fb IS
   (component_type , tool ),
   (tool_type , filter ),
```

```
   (multiple_threads , true ),
   (mutex_provided , false ),
   (form1 , guarded ),
   (static_size , false ),
   (form2 , unbounded ),
   (garbage_collection_provided , true ),
   (form3 , managed ),
   (iterator_provided , false ),
   (form4 , noniterator );
END init_fb;

IRULE BASE booch_irules IS

-- #######
-- The RBDL is truncated here.  Finally the system prompt returns indicating
-- completion of the test run.

%
```

If there are any errors in the RBDL specification file, they must first be corrected, and the RBDL processor re-executed, before the desired application can be created. The following short transcript, shows the detection of a RBDL error.

```
% rbdl < rbdl.example.bad
Parsing input.
syntax error on line 43 scanning Responses
%
```

Line 43 contains a misspelled keyword. Note that the syntax error processing stops after the first error is detected. The user must correct this first error and then run the RBDL processor again, repeating the "correct and run sequence" until all errors have been removed.

7. Notes

This section presents some basic background material for the AdaTAU component of the Reusability Library Framework. The following sections present the major notions and objects that have contributed to the design of AdaTAU.

7.1. Facts

By *fact*, we mean any dynamic quantum of information that our system must be able to process. Typically, facts are stored in a fairly rigid form that is designed to provide efficient access for the system. Some common organizational schemes are property-boolean state pairs, or attribute-value pairs or more generally, triples denoting object names, attribute names, and corresponding values. For example, we can write <printer_indicator_lite_on, true>, or <printer_indicator_lite, on> or <printer, indicator_lite, on>.

Fact structures can be considerably more complicated. At one extreme, one can imagine English-like clauses, or arbitrary lists that can themselves contain sublists. For example, <father_of, sam, <husband_of sarah>> can be used to represent the fact that sam's father is sarah's husband.

Fact Bases

Collections of facts are called *fact bases*. Fact bases can themselves be organized to aid efficient retrieval and modifications. One common tactic is to provide an indexing scheme so that individual facts can be quickly stored and located.

Several facts can each contribute some incomplete facet of a situation that is to be represented within the fact base. For example, <printer_indicator_lite, on> vs. <printer_indicator_lite, blinking>. On the other hand, two facts can stand in contradiction to one another (<printer_indicator_lite, on> vs. <printer_indicator_lite, off>). Thus any system employing a fact base must be designed to handle related facts, and to deal with contradictory facts. Of course, one method is to simply ignore any fact relationships as well as any fact contradictions.

7.2. Rules

A *rule* is a formalized statement that prescribes how a fact base can be changed based on the current state of the fact base. A very common style in which to specify rules is an *if A — then B* format where A and B are placeholders for one or more facts. For example, we can write

if <watch, gold>
then <watch, expensive>

Such a rule can be said to be *primed* if the facts in the collection A are all currently within the fact base. One possible action within a rule base system is to add all the facts within B to the fact base, after a rule becomes primed. Such a rule is said to *fire*.

Rule Antecedents and Consequents

Those facts that must be present in the fact base before the rule is able to fire are called the *antecedent* facts. Such facts are also called *premises*. Analogously, those facts

which should be added to the fact base by the rule are called the *consequent* facts. Thus, in the *if A — then B* rule formalism, A stands for the antecedent facts, and B stands for the consequents. Consequents are also called *conclusions*. In general, the lists A and B can be broken up further. For example, a rule could state that some of the facts in the A list should be absent in order for the rule to fire. Such facts would be *negative* antecedents. Analogously, some of the facts in the collection B might be identified as facts to remove from the fact base when the rule is fired.

Rule Bases

A complete collection of rules organized to capture knowledge in a particular domain is called a *Rule Base*. Just as in fact bases, collections of rules can be organized for efficient processing. A typical use of a rule base system is to begin with a collection of facts as well as a collection of rules, and then fire the primed rules successively, thereby causing new facts to be added to the fact base (or old facts removed). There are scheduling problems (for example, how to choose which of several primed rules to fire first) and this direct approach of incrementally building up the fact base is not appropriate in all cases.

7.3. Inference

The use of a rule base combined with a fact base to explore and draw conclusions about a particular domain is called *inferencing*. There are two well-known inference strategies for rule base systems.

Forward-Chained Inference

Forward-chained inference occurs when the consequent facts of a primed rule are automatically added to the fact base upon the firing of the rule. Forward-chained systems typically are used in a stand-alone fashion where an initial fact base is presented to the inference system along with a collection of rules. The system then runs without explicit user interaction and the fact base is updated as a result of applying rules within the rule base successively to the fact base.

In most cases, a fact should be represented in the fact base only once, so that consequents should be added only if they are not already in the fact base. However, it is useful to store an indication that more than one rule has lead to a particular fact being present in the fact base (see the section on truth maintenance). When no un-primed rules remain, further progress is not possible unless the user can be consulted to add new facts directly.

If more than one rule is found to be primed, some decision must be made about which rule is to be fired first. If rules are stored and examined in some fixed order, one strategy is to simply fire the first primed rule found and go on to the next. Note that in the simplest case where new facts are simply being added to the fact base, a potential live-lock situation is encountered. Once a rule is primed, it will always remain primed so that it can potentially fire repeatedly. One restrictive way to handle this case is to mark a rule as "fired" so as to prevent its re-firing. However, there are circumstances where at least limited rule re-firings might be meaningful, especially rules involving facts which refer to variable quantities. Such variables can have different values over time so that rules which refer to them may be initially unprimed, but may be primed later when a

variable's value changes. The use of rules which refer to variables also introduces the necessity for truth maintenance (see below).

Another useful strategy is to assign weights (e.g., integer values) to rules and to examine them in decreasing order according to their weight. Rules with high weights are understood to be important rules which should be examined early and fired if possible. All rules could first be ranked by weight, then examined sequentially, with any fired rules removed from consideration.

After one pass through the rules (weighted or not), the addition of rule consequent facts may cause other rules to become primed. Thus several passes through the rule list may be necessary before no further progress can be made.

Backward-Chained Inference

Backward-chaining occurs when a rule is examined back to front; i.e., the consequent fact(s) is(are) checked against the fact base first. Backward-chaining is often used when the inference process is directed via interaction with a user who states a fact that she or he wishes to deduce. The user has thus posed a *query* to the inference system.

The system, after first checking for the occurrence of the desired fact in the fact base, will look for rules in which the desired fact is present as a consequent. If such a rule is found, the antecedents of this rule generate new queries for facts that must be verified subsequent to the answering of the initial query. These facts are checked analogously to verify them according to the current state of the fact base. If all of the initial query's antecedent facts are found, the desired fact has been verified and the user's query has been answered. If working backwards in this fashion does not answer the query, other rules in the rule base are examined in search of another one whose consequents include the desired fact and the deduction process continues. After no more candidate rules can be found, the inference process ends with the result that the desired fact could not be verified.

Note that the consequent(s) of rules which are verified as part of this process are typically not added explicitly to the fact base. However, backward-chaining systems can be programmed to run in a stand-alone fashion and in this case, intermediate facts can be added to the fact base. One modified form of a general backward-chaining strategy is to first look for rules whose antecedents are all *initially* present in the fact base. Here an initial forward-chaining inference pass may be followed by succeeding passes which operate in a direct backward-chaining fashion. Once again, rules can be weighted so that more "important" rules may be checked early in the inference process.

Monotonic vs. Non-Monotonic Inference

In the previous discussion, fact processing with rules was limited to the addition of new facts to the fact base. Such an inference system is called a *monotonic* inference system. The fact base under a monotonic inferencing package can only grow larger.

In many applications that must model real world representation and manipulation of information, there is a need to handle the deletion (and modification) of facts as well as their addition. Inference systems which support the retraction of information are called *non monotonic* systems. Rules can themselves be equipped with *retract lists* as part of the consequent clause of a rule as well as *assert lists* which contain those facts which are

to be added to the fact base. In this case, if the rule is primed, the members of the retract list must be deleted from the fact base.

Truth Maintenance

When facts are deleted, the effect of deletions can ripple through the fact base with the effect that the conclusions of rules which were fired because of the presence of certain premise facts can now be considered to be invalid. The facts added to the fact base as a result of these now invalidated rules should themselves be withdrawn. This process continues with several passes through the fact base necessary to bring the fact base into a consistent state.

Truth Maintenance is that part of the inferencing system that manages the consistency of the information within the fact base. An elementary example of truth maintenance is a check that for single-valued attributes, only a single fact using this attribute can be part of the fact base at any given time. For example, both of the facts <printer_indicator_lite, on> and <printer_indicator_lite, off> cannot be simultaneously part of the fact base. Multi-valued attributes must still be permitted. Strictly monotonic inferencing systems essentially require no truth maintenance component, and depending on the nature and generality of the fact base, non-monotonic inferencing systems can require very complicated truth maintenance subsystems. For example, in the non-monotonic version of AdaTAU, asserting facts regarding single-valued attributes should automatically trigger the retraction of any earlier value for such attributes; i.e., the value must be updated. Also, facts asserted using rules that refer to variables may need to be withdrawn as a consequence of a change to the variable's value.

7.4. GLOSSARY

Agenda -- An agenda is a weighted list of pending questions the system desires the user to answer. Such questions involve facts that the system cannot deduce directly.

Antecedent – The antecedent refers to the "if" part of a rule.

Ask – That part of AdaTAU which asks questions. The questions are taken from the agenda.

Assert Lists – An assert list is the list of facts, in the consequent of a rule, which are to be added to the fact base when the rule fires.

Backward-Chaining -- Backward-chaining is a form of inferencing. With backward chaining, the rule containing a consequent to be proven is examined. If all the premises in the antecedent are true, the consequent has been proven. If all the premises are not true, then rules which have those premises as consequents are examined. And so on.

Conclusions – The consequents of a rule are also referred to as conclusions.

Consequent – The consequent refers to the "then" part of a rule.

Fact – A fragment of knowledge, represented in a standard form. The pair <compiled,yes> might represent that a code unit being tested had been successfully compiled.

Fact Base – A collection of facts.

Fire – A primed rule is said to fire, when its consequents are added to the fact base.

Forward-Chaining – Forward-chaining is a form of inferencing. With forward-chaining, all rules are examined in turn. If a rule is found which has all of its premises true, it is fired and the resulting consequent facts are added to the fact base. This sequence is repeated until no more rules can fire.

FRules – FRules, or Focus Rules, are used to guide the focuser on its tour of investigators.

Inferencing – The process of using existing facts and existing rules to deduce new facts is called inferencing.

Investigators – Investigators are the individual components of AdaTAU. Each investigator can be viewed as a miniature expert system.

IRules – IRules, or inference rules, are those rules whose consequent contains facts to be added to the fact base.

Primed – When all of the conditions in the antecedent of a rule are true, the rule is said to be primed.

Premise – A fact which corresponds to the antecedent of a rule is called a premise of the rule.

QRules – QRules, or question rules, are those rules whose consequent contains questions to be added to the agenda.

Query – A question posed by the user to the system. A query is often associated with a backward chaining system.

Monotonic – A monotonic system is one where facts may only be added to the fact base.

Non-Monotonic – A non-monotonic system is one where facts may be deleted from, as well added to, the fact base.

Retract Lists – A retract list is the list of facts, in the consequent of a rule, which are to

be deleted from the fact base when the rule fires.

Rule – A rule describes when the fact base may change. When the conditions in the "if" part of the rule are true, the "then" part of the rule is added to the fact base.

Rule Base – A collection of rules is referred to as a rule base.

Think – That part of AdaTAU which processes rules. It decides which facts are to be added to the fact base and which questions are to be added to the agenda.

Truth Maintenance – When a fact is deleted from a fact base, in a non-monotonic system, other facts which may have depended on that fact must also be deleted. This process of keeping the fact base consistent is called truth maintenance.

Update – That part of AdaTAU which processes the answers to questions and adds whatever new facts that result from the questions to the fact base.

APPENDIX A: RBDL Syntax and Summary

This appendix contains a description of the Rule Base Definition Language (RBDL). After an overview of the BNF variant used to describe RBDL, individual language features are presented syntactically, with each syntactic description followed by a short summary of the semantics of each feature. Following the description of the individual features, the appendix closes with a complete syntactic summary and an extended example.

A.1. Extended BNF (EBNF) Meta-Symbols

The syntax of the language is described using an extended BNF. The notation used is the same as the notation used throughout the Ada LRM. A brief description is given below. For a complete description see section 1.5 of the LRM.

- `lower_case_word`
  nonterminal (e.g. `adanet_spec`).
- **`bold_face_word`**
  language token (e.g. **`begin`**).
- `{item}`
  braces enclose item which may be repeated zero or more times.
- `[item]`
  brackets enclose optional item.
- `item1 | item2`
  alternation; either item1 or item2

A.2. RBDL EBNF and Semantics

A.2.1. AdaTAU Specification

Syntax

```
adatau_spec ::=
      local_inferencer_definition
```

Semantics

RBDL is an application specific language used to generate the source for an Ada software component called a *distributed inferencer base*. A distributed inferencer base consists of one or more inferencers, which under control of an external application program, operate on local fact bases and communicate through the use of declared fact parameters. All fact parameters are consistent with a single global fact base schema.

The input to the RBDL processor is an AdaTAU spec. A local AdaTAU spec defines a fact base schema, the initial value of the shared fact base, a set of rule bases, an optional fact parameter declaration that describes how information is to be shared among inferencers and an inferencer.

A fact base schema definition determines the set of valid facts. All facts introduced by the initial fact base definition and the rule base definitions must be consistent with the fact base schema. Fact parameters must also be consistent with a corresponding fact base schema.

A.2.2. Local Inferencer Definition

Syntax

```
local_inferencer_definition ::=
      factbase_schema_def
      [local_inference_schema_def]
      initial_factbase_def
      rule_base_definition_part
      inferencer_def

local_inference_schema_def ::=
      fact parameters is
         [import_list] [export_list]
      end fact parameters ;

export_list ::=
      exports : ( param_description_list ) ;

import_list ::=
      imports : ( param_description_list ) ;

param_description_list ::=
        param_description
      | param_description , param_description_list

param_description ::=
        identifier => optional
      | identifier => mandatory
      | identifier => default
      | identifier => focal

rule_base_definition_part ::= {rule_base_definition}

rule_base_definition ::=
        irulebase_def
      | question_base_def
      | qrulebase_def
      | frulebase_def
```

Semantics

A local fact base schema establishes the shape of facts that may be installed in the local fact base. If the local inferencer is to participate in a distributed set of inferencers, it communicates with other inference bases through the use of fact parameters. Incoming facts correspond to fact attributes listed in the import list. Outgoing fact possibilities are identified by fact attributes on the export list. Fact parameters are classified according to their assumed behavior. Optional parameters are those attributes for which the inferencer to which focus has passed may (but need not) receive a value from the previous inferencer. Mandatory parameters must have corresponding values when entry into the destination inferencer occurs. (Note that the initial fact base may be used to provide values for such mandatory fact attributes.) Default parameters will be equipped with a value when the destination inferencer is reached even if the previous inferencer did not

pass a value. Focal parameters provide values associated with the act of performing an inference context switch. Such values may override facts that were present in the fact base of the previous inferencer.

The initial value of a local fact base is given by a fact base definition. The inference rules and the question rules which are applied by the inferencers are defined in the rule base definition part. A rule base is a named collection of either irules, frules or qrules. An inferencer has access to at most one of each type of rule base.

The definition of a qrule base may be preceeded by the the definition of a question base. A question base contains the questions which are asked during the processing of qrules. Each question in a question base has a name. Each qrule base is associated with exactly one question base, however multiple qrule bases may be associated with the same question base. Each qrule in a qrule base explicitly names a question in the associated question base. If an AdaTAU spec defines any qrule bases, then there must be at least one question base defined. If a qrule base is not preceded by a question base, then it must be associated with a question base introduced by another qrule base definition.

A.2.3. Fact Base Schema Definition

Syntax

```
factbase_schema_def ::=
    fact base schema identifier is
         fact_schema_def
        {fact_schema_def}
    end [identifier] ;

fact_schema_def ::=
    attribute_name_list : attribute_type [attribute_value_list] ;

attribute_name_list ::= attribute_name {, attribute_name}

attribute_type ::= some_of | one_of | any | reference

attribute_value_list ::= ( attribute_value {, attribute_value} )
```

Semantics

A fact base schema is a collection of fact schemas. A fact schema defines a set of values which can be associated with an attribute name.

If the attribute name list contains more than one name, it is equivalent to a sequence of fact schema definitions, such that each definition contains a single attribute name from the original list. An attribute name may only appear once in a fact base schema definition.

If the reserved word SOME_OF or ONE_OF appears in the definition then an attribute value list must be provided. If the reserved word ANY appears in the definition then an attribute value list is not permitted. Likewise, if the reserved word REFERENCE appears in the definition then the corresponding attribute names refer to files that contain the complete text of the corresponding "fact value". No corresponding value list is permitted in this case as well.

The fact base schema is used to check the validity of facts introduced by other definitions. Any fact which is used in an AdaTAU specification must be consistent with the fact base schema in

the following ways: (1) the fact's attribute name must be in the fact base schema, and (2) the fact's value must be one of the allowable values defined by the fact schema definition for the attribute name. If an identifier appears at the end of a fact base schema definition, it must repeat the fact base schema name.

A.2.4. Factbase Definition

Syntax

```
initial_factbase_def ::=
   initial fact base identifier is
      fact_list ;
   end [identifier] ;

fact_list ::= null | fact {, fact}
```

Semantics

An initial fact base definition defines an initial value of the local fact base. All facts in the fact list are checked for consistency with the fact base schema. If an identifier appears at the end of a fact base definition, it must repeat the fact base name.

A.2.5. Irulebase Definition

Syntax

```
irulebase_def ::=
    irule base identifier is
       irule_def {irule_def}
    end [identifier] ;

irule_def ::=
    irule identifier is
        antecedent : antecedent_fact_list ;
        consequent : consequent_fact_list ;
        [justification]
    end irule ;

antecedent_fact_list ::= fact {,  fact }

consequent_fact_list ::= gen_fact_list

gen_fact_list ::=
          fact
     |    ~ fact
     |    gen_fact_list , fact
     |    gen_fact_list , ~ fact
```

```
justification ::=
       justification : text_block ;
```

Semantics

An irule base definition defines a named collection of inference rules. Each inference rule is defined by an irule definition. An irule base is used by an inferencer. If an identifier appears at the end of an irule base definition, it must repeat the name of the irule base.

Each fact list in an irule definition must consist of facts which are consistent with the fact base schema.

A.2.6. Questionbase Definition

Syntax

```
question_base_def ::=
   question base identifier is
      question_def {question_def}
   end [identifier] ;

question_def ::=
    question identifier is
       text : text_block ;
       type : quest_attribute_type ;
       [possible_responses]
    end question ;

quest_attribute_type ::= some_of | one_of | any

possible_responses ::=
     responses : response_list

response_list ::= response {response}

response ::=
    response_display {| response_display}
           => gen_fact_list ;

response_display ::= string
```

Semantics

A question base definition defines a named collection of questions which may be referenced by the qrules in a qrule base. If an identifier appears at the end of a question base definition, it must repeat the question base name.

A question definition defines a question and introduces a name for it. Each fact in the response assert fact list must be consistent with the fact base schema.

A.2.7. Qrulebase Definition

Syntax

```
qrulebase_def ::=
    qrule base identifier ( question_base_identifier ) is
       qrule_def {qrule_def}
    end [identifier] ;

qrule_def ::=
   qrule identifier is
       antecedent : antecedent_fact_list ;
       question   : question_identifier ;
       weight     : weight ;
       [justification]
   end qrule ;

question_base_identifier ::= identifier

question_identifier ::= identifier

weight ::= number
```

Semantics

A qrule base definition defines a named collection of question rules. A qrule base definition names the question base it is associated with. Each qrule must name a question defined in this base. If an identifier appears at the end of an qrule base definition, it must repeat the name of the qrule base. Each qrule in a qrule base is defined by a qrule definition.

A.2.8. Frulebase Definition

Syntax

```
frulebase_def ::=
    frule base identifier is
        frule_def {frule_def}
    end [identifier] ;

frule_def ::=
    frule identifier is
        antecedent    : antecedent_fact_list ;
        export        : export_fact_list ;
        focus         : inferencer_id ;
        weight        : weight ;
        [justification]
    end frule ;

inferencer_id ::= identifier
```

```
export_fact_list ::= fact_list
```

Semantics

The inferencer_id named in an frule must correspond to an inferencer defined in another RBDL inferencer specification. Facts in the export_fact_list are added to the fact base just before an inference context switch is processed to support fact parameter processing. A frule base definition defines a named collection of focus rules. If an identifier appears at the end of an frule base definition, it must repeat the name of the frule base. Each frule in a frule base is defined by a frule definition.

A.2.9. Inferencer Definition

Syntax

```
inferencer_def ::=
    inferencer identifier is
         [irule_base_specification]
         [qrule_base_specification]
         [frule_base_specification]
     end [identifier] ;

irule_base_specification ::=
      irule base : ident ;

qrule_base_specification ::=
      qrule base : ident ;

frule_base_specification ::=
      frule base : ident ;
```

Semantics

An inferencer definition defines an inferencer. It specifies the name of the irule base, frule base and the qrule base to be used. If an identifier appears at the end of an inferencer definition, it must repeat the name of the inferencer.

A.2.10. Fact

Syntax

```
fact ::= ( attribute_name , attribute_value )

attribute_name ::= identifier | string

attribute_value ::= identifier | string | number
```

Semantics

A fact consists of an attribute name and an attribute value. The attribute name must appear in the fact base schema. The attribute value must be an allowable value for the attribute name, as defined by the fact schema for the attribute name.

The actual representation of an attribute name and value is a string. For convenience identifiers and numbers are also allowed syntactically. If the alternate forms (identifier or number) are given, they are converted to strings by the RBDL processor, enclosing the actual text in quotes.

A.2.11. Lexical Elements

Syntax

```
identifier ::= letter {[underline] letter_or_digit}

letter ::= upper_case_letter | lower_case_letter

number ::= digit {digit}

string ::= "{graphic_character}"

text_block ::= left_brace {graphic_character} right_brace
```

Semantics

Identifiers, numbers and strings must be fully contained on a single line. Text_blocks are allowed to span multiple lines. Moreover, identifiers and number must be separated from each other by at least one separator. A separator is either a space character, a tab character, or an end of line.

A.3. RBDL EBNF Syntax Summary

The following is a summary of the EBNF description of the RBDL syntax. Terms are introduced in depth-first fashion.

```
adatau_spec ::=
      local_inferencer_definition

factbase_schema_def ::=
    fact base schema identifier is
         fact_schema_def
        {fact_schema_def}
    end [identifier] ;

fact_schema_def ::=
     attribute_name_list : attribute_type [attribute_value_list] ;

attribute_name_list ::= attribute_name {, attribute_name}

attribute_type ::= some_of | one_of | any | reference

attribute_value_list ::= ( attribute_value {, attribute_value} )
```

```
local_inferencer_definition ::=
       factbase_schema_def
       [local_inference_schema_def]
       initial_factbase_def
       rule_base_definition_part
       inferencer_def

local_inference_schema_def ::=
      fact parameters is
         [import_list] [export_list]
      end fact parameters ;

export_list ::=
      exports : ( param_description_list ) ;

import_list ::=
      imports : ( param_description_list ) ;

param_description_list ::=
        param_description
      | param_description , param_description_list

param_description ::=
        identifier => optional
      | identifier => mandatory
      | identifier => default
      | identifier => focal

rule_base_definition_part ::= {rule_base_definition}

rule_base_definition ::=
        irulebase_def
      | question_base_def
      | qrulebase_def
      | frulebase_def

initial_factbase_def ::=
    initial fact base identifier is
       fact_list ;
    end [identifier] ;

irule_def ::=
     irule identifier is
         antecedent : antecedent_fact_list ;
         consequent : consequent_fact_list ;
         [justification]
     end irule ;

antecedent_fact_list ::= fact {,  fact }

consequent_fact_list ::= gen_fact_list
```

```
gen_fact_list ::=
          fact
     |    ~ fact
     |    gen_fact_list , fact
     |    gen_fact_list , ~ fact

weight ::= number

justification ::=
       justification : text_block  ;

fact_list ::= null | fact {, fact}

irulebase_def ::=
     irule base identifier is
        irule_def {irule_def}
     end [identifier] ;

question_base_def ::=
    question base identifier is
       question_def {question_def}
    end [identifier] ;

question_def ::=
     question identifier is
        text : text_block ;
        type : quest_attribute_type ;
        [possible_responses]
     end question;

quest_attribute_type ::= some_of | one_of | any

possible_responses ::=
      responses : response_list

response_list ::= response {response}

response ::=
     response_display {| response_display}
            => gen_fact_list ;

response_display ::= string

qrulebase_def ::=
     qrule base identifier ( question_base_identifier ) is
        qrule_def {qrule_def}
     end [identifier] ;

question_base_identifier ::= identifier
```

```
qrule_def ::=
    qrule identifier is
        antecedent : antecedent_fact_list ;
        question   : question_identifier ;
        weight     : weight ;
        [justification]
    end qrule ;

question_identifier ::= identifier

frulebase_def ::=
     frule base identifier is
        frule_def {frule_def}
     end [identifier] ;

frule_def ::=
   frule identifier is
        antecedent    : antecedent_fact_list ;
        export        : export_fact_list ;
        focus         : inferencer_id ;
        weight        : weight ;
        [justification]
    end frule ;

export_fact_list ::= fact_list

inferencer_id ::= identifier

inferencer_def ::=
    inferencer identifier is
       [irule_base_specification]
       [qrule_base_specification]
       [frule_base_specification]
    end [identifier] ;

irule_base_specification ::=
       irule base : ident ;

qrule_base_specification ::=
       qrule base : ident ;

frule_base_specification ::=
       frule base : ident ;

fact ::= ( attribute_name , attribute_value )

attribute_name ::= identifier | string

attribute_value ::= identifier | string | number

ident ::= identifier

identifier ::= letter {[underline] letter_or_digit}

letter ::= upper_case_letter | lower_case_letter
```

```
number ::= digit {digit}
string ::= "{graphic_character}"
text_block ::= left_brace {graphic_character} right_brace
```

APPENDIX B: RBDL Extended Example

```
-- -------------------- Disclaimer ----------------------------------
--
-- This software and its documentation are provided "AS IS" and
-- without any expressed or implied warranties whatsoever.
-- No warranties as to performance, merchantability, or fitness
-- for a particular purpose exist.
--
-- In no event shall any person or organization of people be
-- held responsible for any direct, indirect, consequential
-- or inconsequential damages or lost profits.
--

Fact base Schema Booch_Facts is

            component_type : one_of (structure, tool, subsystem);

            granularity : one_of (monolithic, polylithic);

            structure_type : one_of
                    (stack, queue, string, deques, rings, maps,
                     sets, bags, lists, trees, graphs, other);

            tool_type : one_of
                    (utility, filter, pipe, sorting, searching,
                     pattern_matching, other);

            determine_basic_props : one_of (yes, no);
            multiple_threads,
            mutex_provided,
            multiple_readers,
            static_size,
            garbage_collection_provided,
            controlled,
            iterator_provided: one_of (true, false);

            form1 : one_of (sequential, guarded, concurrent, multiple);

            form2 : one_of (unbounded, bounded);

            form3 : one_of (unmanaged, managed, controlled);

            form4 : one_of (iterator, noniterator);

End Booch_Facts;

INITIAL FACT BASE init_fb IS
   (determine_basic_props , yes);
END init_fb;


Question base Booch_Questions is

    Question Ask_Component_Type is
      Text: (What is the component type?);
```

```
  Type: one_of;
  Responses:
          "structure" => (component_type, structure);
          "tool" => (component_type, tool);
          "subsystem" => (component_type, subsystem);
End question;

Question Ask_Structure_Type is
  Text: {What is the structure?};
    Type: one_of;
    Responses:
          "stack" => (structure_type, stack), (granularity, monolithic);
          "string"=> (structure_type, string), (granularity, monolithic);
          "queue" => (structure_type, queue), (granularity, monolithic);
          "deque" => (structure_type, deque), (granularity, monolithic);
          "ring"  => (structure_type, ring), (granularity, monolithic);
          "map"   => (structure_type, map), (granularity, monolithic);
          "set"   => (structure_type, set), (granularity, monolithic);
          "bag"   => (structure_type, bag), (granularity, monolithic);
          "list"  => (structure_type, list), (granularity, polylithic);
          "tree"  => (structure_type, tree), (granularity, polylithic);
          "graph" => (structure_type, graph), (granularity, polylithic);
          "other" => (structure_type, other);
End question;

Question Ask_Granularity is
  Text:{Is the structure};
  Type: one_of;
  Responses:
     "monolithic (parts are not individually accessible)"
                 => (granularity, monolithic);
     "polylithic (parts are individually accessible)"
                 => (granularity, polylithic);
End question;

Question Ask_Tool_Type is
  Text: {Is the tool a};
  Type: one_of;
  Responses:
          "utility" => (tool_type, utility);
          "filter" => (tool_type, filter);
          "pipe" => (tool_type, pipe);
          "sorting tool" => (tool_type, sorting);
          "searching tool" => (tool_type, searching);
          "pattern matching tool" => (tool_type, pattern_matching);
          "other" => (tool_type, other);
End Question;

Question Ask_About_Mult_Threads is
  Text: {Are the semantics of the component preserved when
        more than one thread of control exists?};
  Type: one_of;
  Responses:
          "yes" => (multiple_threads, true);
          "no" => (multiple_threads, false);
End Question;
```

```
    Question Ask_About_Mut_Exclusion is
      Text: {Does the component guarantee mutual exclusion, or does
             the user have to ensure mutual exclusion};
      Type: one_of;
      Responses:
               "mutual exclusion guaranteed" => (mutex_provided, true);
               "user must provide mutual exclusion" => (mutex_provided, false);
    End Question;

    Question Ask_About_Mult_Readers is
      Text: {Is it possible for more than one reader to have
             simultaneous access to an object?};
      Type: one_of;
      Responses:
               "yes" => (multiple_readers, true);
               "no" => (multiple_readers, false);
    End Question;

    Question Ask_About_Size is
      Text: {Do objects that are part of the component have a size
             that is static throughout the object's life, or does
             the size change dynamically?};
      Type: one_of;
      Responses:
               "size is static" => (static_size, true);
               "size changes dynamically" => (static_size, false);
    End Question;

    Question Ask_About_Gbg_Collection is
      Text: {Does the component take care of garbage collection?};
      Type: one_of;
      Responses:
               "yes" => (garbage_collection_provided, true);
               "no" => (garbage_collection_provided, false);
    End Question;

    Question Ask_About_Controlled_GC is
      Text: {Is garbage collection provided even when multiple
             tasks are accessing different objects in the component?};
      Type: one_of;
      Responses:
                "yes" => (controlled, true);
                "no" => (controlled, false);
    End Question;

    Question Ask_About_Iterators is
      Text: {Is an iterator provided for this component?};
      Type: one_of;
      Responses:
               "yes" => (iterator_provided, true);
               "no" => (iterator_provided, false);
               "not applicable" => (iterator_provided, false);
    End Question;

End Booch_Questions;
```

```
qrule base Booch_Qrules (Booch_Questions) is

    Qrule Get_Component_Type is
        antecedent: (determine_basic_props, yes);
        question: Ask_Component_Type;
        weight : 5;
        justification: {The first set of categories that components are
                        broken down into is structures (data objects or
                        classes of objects), tools (operations or collec-
                        tions of operations to be performed on some
                        structure), and subsystems (a larger abstraction
                        comprising several structures and tools).};
    End Qrule;

    Qrule Get_Structure_Type is
        antecedent: (component_type, structure);
        question: Ask_Structure_Type ;
        weight : 4;
        justification: {Structures are further broken down into these
                        general categories.};
    End Qrule;

    Qrule Get_Granularity is
        antecedent: (structure_type, other);
        question: Ask_Granularity ;
        weight : 4;
        justification: {Structures are also categorized as monolithic
                        or polylithic.};
    End Qrule;

    Qrule Get_Tool_Type is
        antecedent: (component_type, tool);
        question: Ask_Tool_Type ;
        weight : 4;
        justification: {Tools are further broken down into these
                        general categories.};
    End Qrule;

    Qrule Multiple_Threads is
        antecedent: (determine_basic_props, yes);
        question: Ask_About_Mult_Threads;
        weight : 3;
        justification: {Whether or not a component can support multiple
                        threads of control determines whether it is a
                        strictly sequential component or one of the
                        other forms of non-sequential components.  This
                        question helps determine which of the first set
                        of Booch's forms the component falls into.};
    End Qrule;

    Qrule Mutual_Exclusion is
        antecedent: (multiple_threads, true);
        question: Ask_About_Mut_Exclusion;
        weight : 2;
        justification: {If a component supports multiple threads of
                        control, then mutual exclusion becomes an issue.
                        If mutual exclusion is not supported, then the
```

```
                    component of the guarded type.  This question
                    helps determine which of the first set of
                    Booch's forms the component falls into.};
End Qrule;

Qrule Multiple_Readers is
    antecedent: (multiple_threads, true),
                (mutex_provided, true);
    question: Ask_About_Mult_Readers;
    weight : 2;
    justification: {If a component supports mutual exclusion, then
                    the allowance of multiple readers becomes an
                    issue.  If the component does not allow multiple
                    readers, then it is a concurrent component;
                    otherwise it is a multiple component.  This
                    question helps determine which of the first set
                    of Booch's forms the component falls into.};
End Qrule;

Qrule Size is
    antecedent: (determine_basic_props , yes);
    question: Ask_About_Size;
    weight : 1;
    justification: {The second set of Booch's forms is concerned
                    with whether or not the objects in the component
                    are of bounded size.  This question determines
                    whether the component is bounded or unbounded.};
End Qrule;

Qrule Garbage_Collection is
    antecedent: (static_size, false);
    question: Ask_About_Gbg_Collection;
    weight : 1;
    justification: {The third set of Booch's forms is concerned
                    with whether or not garbage collection is
                    provided.  This question determines whether the
                    component is managed or unmanaged.};
End Qrule;

Qrule Controlled_GC is
    antecedent: (multiple_threads, false),
                (static_size, false),
                (garbage_collection_provided, true);
    question: Ask_About_Controlled_GC;
    weight : 1;
    justification: {If the component is sequential (supporting
                    only one thread), then the issue of whether or
                    not garbage collection is provided even when
                    multiple tasks are accessing different objects
                    within the component.  If so, then the component
                    is controlled.};
End Qrule;

Qrule Iterators is
    antecedent: (determine_basic_props, yes);
    question: Ask_About_Iterators;
    weight : 0;
```

```
        justification: {The fourth set of Booch's forms is concerned
                        with whether or not the component provides an
                        iterator for the objects in the component.  This
                        question determines whether the component is an
                        iterator or noniterator component.};
    End Qrule;

End Booch_Qrules;


Irule base Booch_Irules is

    Irule Form1_Sequential is
        antecedent: (multiple_threads, false);
        consequent: (form1, sequential);
        justification: {If the component does not support multiple
                        threads of control, then it is a sequential
                        component.  This is one of the first set of
                        Booch's forms.};
    End Irule;

    Irule Form1_Guarded is
        antecedent: (multiple_threads, true),
                    (mutex_provided, false);
        consequent: (form1, guarded);
        justification: {If the component supports multiple threads of
                        control, but does not provide mutual exclusion,
                        then it is a guarded component.  This is one of
                        the first set of Booch's forms.};
    End Irule;

    Irule Form1_Concurrent is
        antecedent: (multiple_threads, true),
                    (mutex_provided, true),
                    (multiple_readers, false);
        consequent: (form1, concurrent);
        justification: {If the component supports multiple threads of
                        control and mutual exclusion, but does not allow
                        multiple readers simultaneous access, then it is
                        a concurrent component.  This is one of the
                        first set of Booch's forms.};
    End Irule;

    Irule Form1_Multiple is
        antecedent: (multiple_threads, true),
                    (mutex_provided, true),
                    (multiple_readers, true);
        consequent: (form1, multiple);
        justification: {If the component supports multiple threads of
                        control, mutual exclusion, and multiple readers,
                        then it is a multiple component.  This is one of
                        the first set of Booch's forms.};
    End Irule;

    Irule Form2_Bounded is
        antecedent: (static_size, true);
        consequent: (form2, bounded);
```

```
    justification: {If all of the objects associated with the
                    component use the same amount of space
                    throughout their lifecycles (are of static
                    size), then the component is a bounded
                    component.  This is one of the second set of
                    Booch's forms.};
End Irule;

Irule Form2_Unbounded is
    antecedent: (static_size, false);
    consequent: (form2, unbounded);
    justification: {If some of the objects associated with the
                    component change their size during their
                    lifecycle, then the component is an unbounded
                    component.  This is one of the second set of
                    Booch's forms.};
End Irule;

Irule Bounded_Managed is
    antecedent: (form2, bounded);
    consequent: (form3, managed);
    justification: {If the component is bounded, then garbage
                    collection is not an issue, so the component is
                    classified as managed.  This is one of the third
                    set of Booch's forms.};
End Irule;

Irule Sequential_Unmanaged is
    antecedent: (form1, sequential),
                (form2, unbounded),
                (garbage_collection_provided, false);
    consequent: (form3, unmanaged);
    justification: {If a sequential, unbounded component does not
                    provide garbage collection, then it is an
                    unmanaged component.  This is one of the third
                    set of Booch's forms.};
End Irule;

Irule Sequential_Managed is
    antecedent: (form1, sequential),
                (form2, unbounded),
                (garbage_collection_provided, true),
                (controlled, false);
    consequent: (form3, managed);
    justification: {If a sequential, unbounded component does
                    provide garbage collection, but only when one
                    task is accessing the component (not
                    controlled), then it is a managed component.
                    This is one of the third set of Booch's forms.};
End Irule;

Irule Sequential_Controlled is
    antecedent: (form1, sequential),
                (form2, unbounded),
                (garbage_collection_provided, true),
                (controlled, true);
    consequent: (form3, controlled);
```

```
    justification: {If a sequential, unbounded component does
                    provide garbage collection, even when more than
                    one task is accessing different objects within
                    the component, then it is a controlled
                    component.  This is one of the third set of
                    Booch's forms.};
End Irule;

Irule Guarded_Unmanaged is
    antecedent: (form1, guarded),
                (form2, unbounded),
                (garbage_collection_provided, false);
    consequent: (form3, unmanaged);
    justification: {If a guarded, unbounded component does not
                    provide garbage collection, then it is an
                    unmanaged component.  This is one of the third
                    set of Booch's forms.};
End Irule;

Irule Guarded_Managed is
    antecedent: (form1, guarded),
                (form2, unbounded),
                (garbage_collection_provided, true);
    consequent: (form3, managed);
    justification: {If a guarded, unbounded component does
                    provide garbage collection, then it is a
                    managed component.  This is one of the third
                    set of Booch's forms.};
End Irule;

Irule Concurrent_Unmanaged is
    antecedent: (form1, concurrent),
                (form2, unbounded),
                (garbage_collection_provided, false);
    consequent: (form3, unmanaged);
    justification: {If a concurrent, unbounded component does not
                    provide garbage collection, then it is an
                    unmanaged component.  This is one of the third
                    set of Booch's forms.};
End Irule;

Irule Concurrent_Managed is
    antecedent: (form1, concurrent),
                (form2, unbounded),
                (garbage_collection_provided, true);
    consequent: (form3, managed);
    justification: {If a concurrent, unbounded component does
                    provide garbage collection, then it is a
                    managed component.  This is one of the third
                    set of Booch's forms.};
End Irule;

Irule Multiple_Unmanaged is
    antecedent: (form1, multiple),
                (form2, unbounded),
                (garbage_collection_provided, false);
    consequent: (form3, unmanaged);
```

```
        justification: {If a multiple, unbounded component does not
                        provide garbage collection, then it is an
                        unmanaged component.  This is one of the third
                        set of Booch's forms.};
    End Irule;

    Irule Multiple_Managed is
        antecedent: (form1, multiple),
                    (form2, unbounded),
                    (garbage_collection_provided, true);
        consequent: (form3, managed);
        justification: {If a multiple, unbounded component does
                        provide garbage collection, then it is a
                        managed component.  This is one of the third
                        set of Booch's forms.};
    End Irule;

    Irule Form4_Iterator is
        antecedent: (iterator_provided, true);
        consequent: (form4, iterator);
        justification: {If the component provides an iterator for its
                        objects, then it is an iterator component.  This
                        is one of the fourth set of Booch's forms.};
    End Irule;

    Irule Form4_Noniterator is
        antecedent: (iterator_provided, false);
        consequent: (form4, noniterator);
        justification: {If the component does not provide an iterator
                        for its objects, then it is a noniterator
                        component.  This is one of the fourth set of
                        Booch's forms.};
    End Irule;

End Booch_Irules;

inferencer Booch_taxonomy is
   irule base : booch_irules;
   qrule base : booch_qrules;
end Booch_taxonomy;
```

References

[Barr81] A. Barr and E. A. Feigenbaum, *The Handbook of Artificial Intelligence, Volume 1*, William Kaufmann, Inc., 1981.

[Booch87] G. Booch, *Software Components with Ada*, Benjamin/Cummings Publishing Company Inc, Menlo Park, California, 1987.

[LRM83] *Reference Manual for the Ada Programming Language*, United States Department of Defense, February 1983. (American National Standards Institute/MIL-STD-1815A-1983).

[McDowell89] R. McDowell and K. Cassell, "The RLF Librarian: A Reusability Librarian Based on Cooperating Knowledge-Based Systems," *Proceedings of RADC 4th Annual Knowledge-Based Software Assistant Conference*, Utica, NY, September 1989.

[Simos88] M. Simos, "The Growing of an Organon: A Hybrid Knowledge-Based Technology and Methodology for Software Reuse," *Proceedings of 1988 National Institute for Software Quality and Productivity (NISQP) Conference on Software Reusability*, April 1988, pp. E-1 through E-25.

[Solderitsch89] J. Solderitsch, K. Wallnau, and J. Thalhamer, "Constructing Domain-Specific Ada Reuse Libraries," *Proceedings of Seventh Annual National Conference on Ada Technology*, March 1989.

[Wallnau88] K. Wallnau, J. Solderitsch, M. Simos, R. McDowell, K. Cassell, and D. Campbell, "Construction of Knowledge-Based Components and Applications in Ada," *Proceedings of AIDA-88, Fourth Annual Conference on Artificial Intelligence & Ada*, November 1988, pp. 3-1 through 3-21.